Symbolic execution proofs for higher order store programs

Article (Accepted Version)

Noname manuscript No.
(will be inserted by the editor)

Symbolic execution proofs for higher order store programs

Bernhard Reus · Nathaniel Charlton · Ben Horsfall

the date of receipt and acceptance should be inserted later

Abstract Higher order store programs are programs which store, manipulate and invoke code at runtime. Important examples of higher order store programs include operating system kernels which dynamically load and unload kernel modules. Yet conventional Hoare logics, which provide no means of representing changes to code at runtime, are not applicable to such programs. Recently, however, new logics using nested Hoare triples have addressed this shortcoming.

In this paper we describe, from top to bottom, a sound semi-automated verification system for higher order store programs. We give a programming language with higher order store features, define an assertion language with nested triples for specifying such programs, and provide reasoning rules for proving programs correct. We then present in full our algorithms for automatically constructing correctness proofs.

In contrast to earlier work, the language also includes ordinary (fixed) procedures and mutable local variables, making it easy to model programs which perform dynamic loading and other higher order store operations. We give an operational semantics for programs and a step-indexed interpretation of assertions, and use these to show soundness of our reasoning rules, which include a deep frame rule which allows more modular proofs.

Our automated reasoning algorithms include a scheme for separation logic based symbolic execution of programs, and automated provers for solving various kinds of entailment problems. The latter are presented in the form of sets of derived proof rules which are constrained enough to be read as a proof search algorithm.

Keywords Program verification, higher order store, recursion through the store, separation logic, automated verification

1 Introduction

Separation logic [37] is a Hoare-style logic for reasoning about heap-manipulating programs, which extends Hoare logic with connectives and rules for *local reasoning*.

Department of Informatics, University of Sussex, Brighton, United Kingdom E-mail: bernhard@sussex.ac.uk, billiejoecharlton@gmail.com, b.g.horsfall@sussex.ac.uk

Local reasoning allows much simpler proofs of heap-manipulating programs, such as those working with linked lists and trees. As a result, the past decade has seen an explosion of interest in separation logic. Separation logic proofs were initially done by hand, but automated tools soon followed, beginning with Smallfoot [4]. Smallfoot provided (semi-)automatic reasoning about C-like recursive procedures, concentrating on memory safety and simple shape properties. The key idea behind Smallfoot, which has been reused and refined in later tools (see e.g. [17,20,21,30]), is that of *symbolic execution with separation logic* [5].

However, in the original separation logic, and hence all the tools based on it, assertions only allowed one to talk about heaps storing primitive data types such as integers and Booleans, and the code of the program was assumed to be fixed. In Reynolds' seminal paper [37], the treatment of code pointers in separation logic is mentioned as an open problem. Programs which introduce or modify procedures (or code or commands) on the heap at runtime were thus left unaddressed. Yet many interesting kinds of programs fall into this category: "hot update" systems which update code while it runs (e.g. [43]), programs which use dynamic loading and unloading of code (such as the Linux kernel [24]), and programs which perform runtime code generation (e.g. [2]). Heaps which contain procedures (or code or commands) have been called *higher order stores*.

Reynolds' open problem has first been addressed by [33] for machine languages and [36] for a C-like language. After that, following ideas of [26], separation logics with *nested Hoare triples* [40,42,41] have been developed. In these logics, assertions can contain Hoare triples which describe the behaviour of code stored on the program's heap allowing one to reason modularly about higher order store programs.

In this paper we take a natural next step by showing how to combine the symbolic execution idea, which has proved so effective for ordinary procedural programs, with nested Hoare triples. This allows us to build a (semi-)automatic verifier for higher order store programs, which we have named Crowfoot. In doing this, several challenges present themselves. Firstly, because assertions now include triples i.e. specifications, we need to create an automatic prover for entailments between specifications, as well as a prover for entailments between assertions. In existing tools only the latter is needed. Secondly, new symbolic execution rules are required for the statements which make use of the higher order nature of the heap; obviously these rules make use of nested triples. Thirdly, when nested triples are added, one may have both $\forall$ and $\exists$ quantifiers appearing at all nesting levels, and these demand a proper treatment; quantifiers can no longer be sidestepped the way they are in e.g. [5]. Fourthly, one wishes to support deep framing to allow more modular proofs, which means one needs to include support for (at least some uses of) the deep framing operators $\otimes$ and $\circ$ [40,41], which do not appear in logics without nested triples.

Running example. Figure 1 presents an example program that uses stored procedures and involves a generic memoiser. The command $\mathsf{eval}\ [a](\mathsf{e})$ is used to call stored code, more precisely it evaluates the content of cell a and if it points to a function it then applies this function to arguments e. If the number of parameters does not match or no function is stored at a the program evaluation gets stuck.

The safety and correctness of the program in Figure 1 can be expressed within our logic (specifications will be presented in Section 3) and a proof can be found by

the proof search algorithms we implement. The proof is almost automatic, however it requires some *hints* which are the grey shaded parts in the figure. These are annotations for the verifier and are *not* part of the program code. They will be explained in Section 2.3.2.

Our example concerns a recursive implementation *fib* of the Fibonacci function, which makes its recursive calls through the store using code stored at address a. Since the "internal" recursive calls are made via a pointer into the store, we can "hook into" the recursion and provide a memoisation routine *mem*, which also caches these internal calls. This kind of memoisation cannot be implemented for a conventional recursive implementation of the Fibonacci function. Our example is more challenging than the factorial function which is typically used [26,3,16] to illustrate recursion through the store.

The memoisation procedure *mem* first looks to see if it can find a cached result for the given input. If found, then the cached result is given, otherwise the recursive procedure f is used to compute the result, which is then added to the cache. The caching is achieved with the help of an association list library (implementation omitted) which is first initialised in *main* by the call to *load_list_lib*. This results in a set of pointers to procedures that manipulate association lists, which are then passed through to the *useFib* procedure where the memoiser is set up and a Fibonacci number calculated.

The procedure *useFib* first creates a new association list and then stores the *fib* procedure on the heap. Next the memoiser is loaded onto the heap which is set up for the loaded *fib* procedure and the new association list, along with the pointers to the list library code. By providing these arguments at the time of storing the procedure on the heap, they are then fixed by partial application such that running the *stored* version of *mem* (by $\mathsf{eval}\ [a](a, n)$) requires providing just the two remaining arguments. The first is a pointer to itself, which is passed through to the function being memoised where it is used for the recursive calls, and the second argument is the input integer.

A short description of this (semi-)automatic verifier and its logic has appeared in [14]. In this article we give an improved and extended presentation, adding significant amount of detail and some additional features. In detail we additionally provide:

- all the deterministic proof search rules of *Crowfoot* exactly how they have been implemented
- all the high-level rules of separation logic for nested triples which are a minor variation of [41]
- a detailed soundness proof of the logic wrt. high-level core rules
- a short soundness proof of the high-level core rules wrt. a step-indexed model inspired by [7] and different from [41]. The model is also different from the one sketched in [14] allowing one to model some features not originally discussed.
- a number of additional proof hint mechanisms that were needed for verification of a reflective visitor implementation [28].

Structure of this paper The rest of this article is structured as follows. In Section 2 we introduce a language for (annotated) higher order store programs, and a language of assertions for reasoning about them. Unlike the language used in [41], our programming language supports three natural features: fixed recursive procedures,

```
const res;

proc fib(a, n) {
 locals p, q, k;
 if n ≤ 0 then {
  [res] := 0;
  ghost fold $Rel(n, 0)
 } else {
  if n = 1 then {
   [res] := 1;
   ghost fold $Rel(?, ?)
  } else {
   k := n − 2;
   eval [a](a, k); p := [res];
   k := n − 1;
   eval [a](a, k); q := [res];
   [res] := p + q;
   ghost fold $Rel(n, p+q)
  }}}

proc mem(lookupL, addL, createL,
     disposeL, al, f, a, n) {
 locals found, b, v;
 ghost unfold $S(?, ?, ?, ?, ?, ?, ?);
 found := new 0;
 eval [lookupL](al, n, found, res);
 b := [found]; dispose found;
 if b = 0 then {
  ghost fold $S(?, ?, ?, ?, ?, ?, ?);
  eval [f](a, n);
  ghost unfold $S(?, ?, ?, ?, ?, ?, ?);
  v := [res]; eval [addL](al, n, v)
 } else { skip };
 ghost fold $S(?, ?, ?, ?, ?, ?, ?) }

proc useFib(lookupL, addL, createL,
     disposeL) {
 locals al, a, f, n;
 f := new 0;
 al := new 0;
 eval [createL](al);
 [f] := proc fib(_, _) deepframe DeepInv ;
 a := new 0;
 [a] := proc mem(lookupL, addL, createL,
     disposeL, al, f, _, _);
 ghost fold $S(?, ?, ?, ?, ?, ?, ?);
 n := 31337;
 eval[a](a, n);
 ghost unfold $S(?, ?, ?, ?, ?, ?, ?);
 ghost unfold $ListLibWeak(?, ?, ?, ?);
 eval [disposeL](al);
 dispose a; dispose f; dispose lookupL;
 dispose addL; dispose createL;
 dispose disposeL; dispose res
}

proc main() {
 locals lookupL, addL, createL, disposeL;
 lookupL := new 0; addL := new 0;
 createL := new 0; disposeL := new 0;
 call load_list_lib(lookupL, addL,
     createL, disposeL);
 ghost unfold $ListLibStrong(?, ?, ?, ?);
 ghost fold $ListLibWeak(?, ?, ?, ?);
 call useFib(lookupL, addL,
     createL, disposeL)
}

proc load_list_lib(lookupL,
 addL, createL, disposeL) {...}
```

Fig. 1 Our running example program. (*DeepInv* is defined in Figure 4.)

integer variables x, fixed procedure names $\mathcal{F}$, integer literals n, declared constants c

$$
\begin{array}{rcl}
\text{address expr } e_A & ::= & x \mid c \mid x + n \mid x + c \\
\text{value expr } e_V & ::= & n \mid x \mid c \mid e_V \bowtie e_V \qquad \text{where } \bowtie \in \{+, -, \times\} \\
\text{statement } C & ::= & \mathsf{skip} \mid At \mid C; C \mid \mathsf{if}\ e_V \bowtie_c e_V\ \mathsf{then}\ C\ \mathsf{else}\ C \\
 & & \mid \mathsf{while}\ e_V \bowtie_c e_V\ \mathsf{do}\ C \qquad \text{where } \bowtie_c \in \{=, \neq, <, \leq\} \\
\text{argument } t & ::= & x \mid c \\
\text{atomic statement } At & ::= & x := e_V \mid x := [e_A] \mid [e_A] := e_V \mid [e_A] := [e_A] \\
 & & \mid x := \mathsf{new}\ {e_V}^+ \mid \mathsf{dispose}\ e_A \mid \mathsf{call}\ \mathcal{F}(t^*) \\
 & & \mid \mathsf{eval}\ [e_A](t^*) \mid [e_A] := \mathsf{proc}\ \mathcal{F}([t|\_]^*)
\end{array}
$$

Fig. 2 Abstract syntax for program statements.

mutable local variables and partial application. The specification of our running example, the memoiser, is presented in Section 3. Section 4 contains the high-level ("core") rules for reasoning about programs with higher-order store. These are an adaptation of the rules of [40], made to fit our programming language. Their soundness is then shown using a model based on step-indexing in Section 5. This model also allows us to use a mildly simplified rule for recursion through the store. Our automated proof search algorithms are described in Section 6. We present our methods in the form of sets of proof rules which are algorithmic in nature; essentially these rules can be read as a proof search algorithm. We show that these rules are sound by deriving them from those of Section 4. In Section 7 we briefly discuss related work and Section 8 contains a short report about our experience with the Crowfoot tool, a verifier based on the symbolic execution rules presented in this paper. Finally, Section 9 concludes by discussing future work.

2 Programming language and assertion language

2.1 Programming language featuring higher order store

We work with an imperative language with recursive procedures, call-by-value parameter passing, and dynamic memory allocation via a mutable heap supporting address arithmetic and, crucially, higher order store operations. Procedure bodies are program statements, whose abstract syntax is given in Figure 2.

Square brackets are used for dereferencing addresses, so $x := [a]$ *reads* the content at address a into the variable x, whereas $[a] := x$ *stores* the value of x at address a in the heap.

For using the higher-order store, there are two important statement forms. Statements like $[a] := \mathsf{proc}\ \mathcal{F}(x, \_)$ are used to load code onto the heap, optionally instantiating some of the parameters at load-time. In this case, the fixed procedure named $\mathcal{F}$ is stored at address a. $\mathcal{F}$ has two parameters, the first of which is instantiated with x (partial application). The stored procedure will then have arity 1. We use an underscore for those parameters that are to be given at invocation time. As our syntax uses _ to represent arguments not yet "filled in", we can supply any subset of the arguments at load-time, not just initial segments.

set variables α, predicate names P

$$
\begin{array}{lrcl}
\text{element expressions} & e_E & ::= & e_V \mid (e_E{}^+) \mid e_S \\
\text{set expressions} & e_S & ::= & \alpha \mid e_S \cup e_S \mid \{e_E\} \mid \mathrm{proj}_n(e_S) \mid \emptyset \\
\text{behavioural spec.} & B & ::= & \forall [x|\alpha]^*.\, \{P\} \cdot (t^*) \{Q\} \\
\text{content spec.} & \mathscr{C} & ::= & e_V \mid \_ \mid B \\
\text{atomic formula} & A & ::= & e_A \mapsto \mathscr{C}^+ \\
 & & & \mid P(e_V{}^*; e_S{}^*) \mid e_V = e_V \mid e_V \neq e_V \\
 & & & \mid e_E \in e_S \mid e_E \notin e_S \mid e_S \subseteq e_S \mid e_S = e_S \\
\text{spatial conjunction} & \Phi, \Theta, \Upsilon & ::= & A \star \Theta \mid \mathsf{emp} \\
\text{assertion disjunct} & \Psi & ::= & \exists [x|\alpha]^*.\Theta \\
\text{assertion} & P, Q & ::= & \mathit{false} \mid \Psi \vee P
\end{array}
$$

Fig. 3 Abstract syntax for the assertion language

The other important statement is $\mathsf{eval}\ [e_A](t_1, \ldots, t_n)$, which lets us run the code stored in cell e_A with actual parameters $t_1, \ldots, t_n$.

Note that the syntax is slightly restrictive, for example there is no $\div$ operator, procedure arguments can only be variables, and the address expressions limit pointer arithmetic to two simple cases of addition. However, these issues are orthogonal to the focus of this research, which is to support reasoning for higher-order store programs, and the verification system presented should be considered a research prototype.

2.2 Assertion language

The syntax for the assertion language used in our automatic proof search algorithms can be seen in Figure 3. We extend the logic [40] which already uses separation logic primitives: predicate emp for the empty heap, $e_1 \mapsto e_2$ for a one cell heaplet with address e_1 and its content e_2, and $\star$ for adding two heaplets with disjoint addresses. Our extended language allows *nested triples* to appear in assertions, such that we can reason about stored procedures. For example, the assertion

$$x \mapsto \forall a.\, \{a \mapsto \_\} \cdot (a)\, \{a \mapsto \_\}$$

states that the content at address x is a procedure of arity 1 which, for all arguments a, satisfies the given Hoare triple $\{a \mapsto \_\} \cdot \{a \mapsto \_\}$. A further addition to the logic of [40] are the introduction of set and element expressions. Element expressions e_E are one of the following: either the usual (integer) expressions e_V, or tuples of elements expressions, eg. (e_1, e_2, e_3) or set expressions. Set expressions e_S are either: a set variable α, the union of two set expressions, a singleton element set (thus sets can be nested), the n-th projection of a set expression, $\mathrm{proj}_n(e_S)$, or the empty set $\emptyset$. Projections map sets of tuples to sets by lifting the standard projection map π_n to sets in the canonical way, for instance $\mathrm{proj}_2(\{(e_1, e_2, e_3)\}$ equals $\{e_2\}$.

We could easily extend the available operations and relations on sets, adding for example intersections or inequalities of sets.

Figure 3 explains at the syntax for atomic formula A. The first case describes a heaplet (ie. a single cell heap). The single cell has address e_A pointing to a list of contiguous cells (in concrete syntax separated by commas) which are specified using one of the following *content specifiers*: an expression e_V describing some

integer content, an underscore _ specifying that any content will suffice, or a *behavioural specification* B stipulating that the cell contains a procedure satisfying Hoare triple B that also indicates the number and kind of arguments[1]. Other ways to obtain atomic formulae are usages of predicates P with integer and set arguments separated by a comma, comparison between (integer) expressions, and logical expressions involving sets. The latter comprise elementhood test, negated elementhood test, subset relationship between sets, and equality between sets.

Spatial conjunctions allow one to describe larger heaps using repeatedly $A \star \ldots$, terminating this process using the empty heap predicate emp. An assertion then consists of a disjunction of existentially quantified spatial conjunctions.

A key feature of the assertion language in Figure 3 is that only certain kinds of formulae are allowed; the use of logical connectives and quantifiers must follow a particular pattern. Restricting the assertion language like this is a standard technique when building formal verification tools: it increases the degree of automation one can achieve, at the expense of the expressiveness of the specifications one can consider. With the restrictions we adopt in Figure 3 (which are similar to those used in Smallfoot [4], for example) we are able to program an effective automatic entailment prover in a fairly natural way. We do not include spatial implication, i.e.. the so-called "*magic wand*". The main reason is simplicity as first-order separation logic with full usage of magic wand is undecidable. Techniques to circumvent this problem in practice and to support magic wand in (semi-)automated proofs have been suggested by [9,39] and could be also used to extend the Crowfoot logic. A sound and complete proof system for separation logic with magic wand has been recently presented in [31] which "may also serve as practical foundation" (*loc.cit.*) for verifiers.

A formula is called *pure* if it does not use $\mapsto$ and contains either no predicate symbols P, or only predicate symbols defined by pure formulae. Thus pure formulae do not refer to the heap. In Section 4.2, we give a slightly non-standard interpretation to the pure formulae, additionally requiring the heap to be empty; e.g. $x = y$ holds exactly when x and y are equal *and* the heap is empty. We do this (following again [4]) so that our restricted assertion language needs only one kind of conjunction, $\star$; it needs not include $\wedge$.

2.3 Annotated programs

Annotated programs are written using the programming and assertion languages given in the previous sub-sections.

2.3.1 Declarations

An annotated program is a sequence of declarations, which can be of the following kinds:

- Constant definition: const c or const $c = n$
- Abstract predicate declaration: forall P(_*; _*) or forall pure P(_*; _*)
  Predicates declared as abstract may be used in specifications, but have no definition (so they cannot be folded or unfolded). Thus a successful proof shows

[1] Recall that arguments t can be a variable x or constant c.

that the program works for *any* definition of such predicates. This feature provides only a "hint" of second-order logic: universal quantification over predicates is possible if the scope is the entire input file. Abstract predicates can optionally be declared pure, which means they can be duplicated and discarded in assertions. This would not be sound for general abstract predicates.

– Inductive/recursive predicate declaration: $\mathsf{recdef}\ \mathrm{P}(x^*;\alpha^*) := P$
The following, for example, declares a linked list segment predicate that appears frequently in the separation logic literature:

$$\begin{array}{lll} \mathsf{recdef}\ \mathrm{Lseg}(x,y;\alpha) & := & x = y * \alpha = \emptyset \\ & \vee & \exists n,d,\beta.\ x \mapsto d, n * \mathrm{Lseg}(n,y;\beta) * \alpha = \{(d,x)\} \cup \beta \end{array} \quad (1)$$

Predicates are implicitly regarded as pure if their definition does not refer to the heap (as defined in Section 2.2). One can also declare specifications designed for reasoning about recursion through the store, such as

$$\mathsf{recdef}\ \mathrm{R}(x) \quad := \quad x \mapsto \forall a.\, \{\mathrm{R}(x) \star y \mapsto \_\} \cdot (a)\, \{\mathrm{R}(x) \star y \mapsto \_\}$$

The importance of such (mixed variant) recursive predicate definitions will be discussed in detail when the proof rule is discussed that deals with recursion through the store.
The well-definedness of a recursively defined predicate, ie. whether it semantically exists, is not checked. To warrant maximum flexibility, our tool does not guarantee this and leaves the existence proof to the user. Thus, it is the user's responsibility to write recursive predicate declarations that are actually meaningful.[2] This is similar in spirit to the way admissibility was handled in LCF [23] where admissibility itself had to be proved outside the LCF logic.

– Declaration using the invariant extension operator, discussed in more detail in Section 2.4:

$$\mathsf{recdef}\ \mathrm{S}(\mathbf{x}) \quad := \quad \mathrm{R}(\mathbf{y}) \circ \Psi$$

where $\mathbf{y} \subseteq \mathbf{x}$ and $fv(\Psi) \subseteq \mathbf{x}$. This means in particular that Ψ must not introduce new free variables, however it can introduce new existentially quantified variables. The (deep) invariant extension operator $\circ$, presented in [40], has the effect of deeply framing an invariant onto the predicate. This involves adding the invariant Ψ to the definition of R, and further adding the assertion as an invariant to all nested specifications. In short, $\mathrm{S}(\mathbf{x}) = (\mathrm{R}(\mathbf{y}) \otimes \Psi) \star \Psi$.

– (Abstract) procedure declaration:

$\mathsf{proc\ abstract}\ \mathcal{F}(x^*)$	$\mathsf{proc}\ \mathcal{F}(x^*)$
$\forall\ [x\|\alpha]^*.$	$\forall\ [x\|\alpha]^*.$
$\mathsf{pre} : P$	$\mathsf{pre} : P$
$\mathsf{post} : Q$	$\mathsf{post} : Q$
	$\{\quad \mathsf{locals}\ x^*; C \quad \}$

This declares a (possibly abstract) procedure, and associates with it a specification consisting of a pre- and postcondition. Formal parameters are taken

[2] We could implement a semantic checker that verifies that the declaration is actually an instance of a certain pattern that guarantees existence but decided against it for maximum flexibility as our verifier is a research tool.

as implicitly universally quantified[3]. We disallow assignments to formal parameters in the procedure body, so that these have the same meaning in the postcondition as in the precondition. Abstract procedures have a specification but no body, and are useful when we wish to model library procedures for which we know the behaviour but do not know or do not care about the source code.

2.3.2 Hints

To assist the prover in verifying a program, certain hints can be provided. Firstly, while-loops are annotated with an invariant:

$$\text{statement } C \quad ::= \quad \ldots \mid \mathsf{while}\ e_V \bowtie e_V \ \boxed{P}\ \mathsf{do}\ C$$

The next kind are annotations to some of the atomic statements, defined in Section 2.1.

$$\begin{aligned}
\text{atomic statement} \quad At \quad ::= \quad & \ldots \mid \mathsf{call}\ \mathcal{F}(t^*)\ \boxed{\text{inst-hints}^*\ \text{deepframe?}} \\
& \mid \mathsf{eval}\ [e_A](t^*)\ \boxed{\text{inst-hints}^*} \\
& \mid [e_A] := \mathcal{F}([t|_]^*)\ \boxed{\text{deepframe?}} \\
\text{inst-hints} \quad ::= \quad & x = e_V \mid \alpha = e_S \\
\text{deepframe} \quad ::= \quad & \mathsf{deepframe}\ \Psi
\end{aligned}$$

The 'inst-hints' annotation, used with the call and eval statements, provides optional hints on how to instantiate quantified variables over the relevant specification. For example if the specification of the procedure we are going to eval is $\forall a, b, c.\, \{P\} \cdot (a)\, \{Q\}$ we might use hint "$b = e$" which will instantiate b with expression e. With no hints for the variable c, the system will attempt to compute a suitable instantiation. The optional deepframe annotation, which consists of the keyword deepframe followed by an assertion disjunct, allows deep framing of an invariant to take place on the triple being used or stored onto the heap, see Section 2.4.

The final kind of annotations are the *ghost* statements, added to the atomic statements:

$$\begin{aligned}
\text{atomic statement} \quad At \quad ::= \quad & \ldots \mid \boxed{\mathsf{ghost}\ G} \\
\text{ghost statement} \quad G \quad ::= \quad & \mathsf{fold}\ \mathrm{P}((x|?)^*; (\alpha|?)^*)\ \text{inst-hints?} \\
& \mid \mathsf{unfold}\ \mathrm{P}((x|?)^*; (\alpha|?)^*) \\
& \mid \mathsf{split}\ \mathrm{P}\ x\ ((e_V|?)^+) \mid \mathsf{join}\ \mathrm{P}\ x
\end{aligned}$$

These annotations, interspersed with program statements in the procedure bodies, tell the verifier when it is necessary to fold or unfold the user-defined predicates. For example, a standard linked-list definition would need to be unfolded if we want to examine the contents, or traverse the list. The split and join statements are special cases that are explained in the next section.

[3] Only other variables thus need to be explicitly quantified by the user.

2.3.3 List segments

Linked lists are a widely used heap data structure so it is important to be able to reason about list segments. For greater flexibility we work with a general pattern of list segment definitions which can be syntactically recognised. For all such definitions an axiom for joining list segments will be available, and for definitions which are *splittable* (explained shortly) an axiom for splitting list segments is additionally available. These can be used via ghost statements to perform inductive list reasoning in proofs.

Let

$$\mathrm{LsegDefs}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k])$$

be the set consisting of the following (syntactic) predicate definitions:

$$\mathsf{recdef}\ \mathrm{P}(s,t;\alpha) \quad := \quad s = t \star \alpha = \emptyset \quad \vee \quad \left(\begin{array}{l} \exists n, \mathbf{v}, \beta\ . \\ \quad s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \quad \star\ A_1 \star \cdots \star A_N \\ \quad \star\ \mathrm{P}(n,t,\beta) \\ \quad \star\ \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \end{array} \right)$$

where:

- $\mathscr{C}_1, \ldots, \mathscr{C}_M$ are content specifiers whose free variables come from $\mathbf{v}$.
- $A_1, \ldots, A_N$ are atomic formulae whose free variables come from $\mathbf{v}$.
- $E_1, \ldots, E_k$ are value expressions whose free variables come from $\mathbf{v}, s$.
- All variables appearing are distinct.

A list segment predicate as above is called *splittable* if one of $E_1, \ldots, E_k$ is either the variable s (which is used as the address of the first list node in the segment), or a variable $v \in \mathbf{v}$ such that one of $A_1, \ldots, A_N$ has the form $v \mapsto \mathscr{C}'$. Having one of these constraints ensures that each element in the abstract set α can be uniquely identified.

Let us demonstrate this with the list segment predicate we saw earlier (1). That definition is in $\mathrm{LsegDefs}(n, d, [d], [], [d, x])$ and is splittable. Thus the ghost statement "split Lseg a (x, y)" is available, and will split a list segment of shape Lseg, which starts at address a, to break out the element (x, y). Thus, the list is split into three parts: the list segment of all elements preceding element (x, y), the element (x, y), and the list segment of the succeeding elements. Statement "$\mathsf{ghost\ join}$ Lseg e" is for concatenating two such segments, taking $\mathrm{Lseg}(e_1, e, e_\alpha) \star \mathrm{Lseg}(e, e_2, e_\beta)$ and producing a joint list $\mathrm{Lseg}(e_1, e_2, e_\alpha \cup e_\beta)$. The corresponding deterministic proof rules (GhostJoin,GhostSplit) are in Figure B.2.

2.4 Deep framing

The *deep frame rule* [8,40] allows one to infer $\{P\}\, C\, \{Q\} \otimes I$ from $\{P\}\, C\, \{Q\}$, where $\otimes$ is a deep framing operator. Intuitively this operator adds the invariant I not just to the pre- and postconditions of the triple $\{P\}\, C\, \{Q\}$, but also to all triples nested *inside* P and Q, at all levels. For example,

$$\begin{array}{rl} & \forall x.\, \{a \mapsto \{\mathsf{emp}\} \cdot ()\, \{\mathsf{emp}\}\} \cdot (x)\, \{\mathsf{emp}\} \otimes y \mapsto \_ \\ \Leftrightarrow & \forall x.\, \{x \mapsto \{y \mapsto \_\} \cdot ()\, \{y \mapsto \_\} \star y \mapsto \_\} \cdot (x)\, \{y \mapsto \_\} \end{array}$$

as can be proved using the distribution laws for $\otimes$ found in [40]. This is useful for modular reasoning as explained in [8] and further demonstrated by our running example. The operator $\circ$ from [40], used in recdef definitions, is a convenient shorthand: $\Phi \circ I := (\Phi \otimes I) \star I$.

The annotation deepframe I prompts the verifier to add the invariant I deeply onto the triple for a procedure; this can be done when a procedure is called with call, or when a procedure is first written to the heap. Note that deep framing is not available for the eval statement, as this would be unsound [15].

Our proof rules implement deep framing using the $\otimes$ distribution laws from [40]. In Section 5 we will show how to distribute $\otimes$ through recursively defined predicates. In our verification system we currently support this distribution operator only for specific recursive definitions of the form

$$\mathrm{R}(\mathbf{x}) \quad := \quad \bigstar_{i=1}^{n} v_i \mapsto \forall \mathbf{a}_i.\, \{\mathrm{R}(\mathbf{e}) \star F_i\} \cdot (\mathbf{p}_i)\, \{\mathrm{R}(\mathbf{e}) \star G_i\} \star H$$

where: $\mathbf{e}$ may contain variables $\mathbf{a}_i$ as well as $\mathbf{x}$, and each F_i, each G_i and H are all left zeroes of $\otimes$ (i.e. informally they do not contain any nested triples). This form is sufficient to cover all the cases we have encountered so far.

3 Specification of the running example

The specifications of the procedures in Figure 1 can be found in Figure 5. The auxiliary predicate definitions are given in Figure 4.

The *fib* implementation. Let us first examine how to specify the *fib* code. Predicate $\$Rel(n, m)$ says that n and m are appropriately related for the function being computed; in this case we define $\$Rel(n, m)$ to mean that m is the nth Fibonacci number. But this definition is only used inside the proof of *fib*, and not when proving the generic components such as *mem* so the proof is *modular*. To emphasise this we could verify the memoiser with a generic specification $\$Rel$ using declaration forall pure $\$Rel(n, m)$. Currently there is no means of making that "forall pure" have a scope other than global (wrt. the input file), so one cannot easily prove that the memoiser satisfies a generic specification *and* verify a particular client in the *same* file. But one can do both separately.

Suppose we try to write a precondition for the *fib* code. This precondition must mention all the heap resources needed by *fib*. Firstly a cell $res \mapsto \_$ is needed into which we write the result (recall that *res* is a global constant). Secondly, since *fib* makes its recursive call through the heap at the address given by parameter a, the precondition must include $a \mapsto B$ where B is a nested triple. In particular, B must state that the code stored at a has the same kind of behaviour as we specify for the *fib* procedure. But we do not have *fib*'s specification yet, because we are still trying to formulate its precondition! It appears that we need a specification which depends on itself. Using the recdef keyword we can declare such a recursively defined specification, namely the $\$RecFn$ predicate, which appears nested inside its own definition.

The memoiser. The memoiser implementation uses an association list data structure, at address al, to cache the input-output pairs for the function being memoised. An association list with a header cell, starting at address al and containing values for a set κ of keys, is described by $\$AssocListH(al; \kappa)$. Such

$$\textsf{recdef}\ \$Rel(n,m) := \quad n \le 0 \star m = 0 \quad \vee \quad n = 1 \star m = 1$$
$$\vee \quad \exists a, b.\ 2 \le n \star \$Rel(n-2,a) \star \$Rel(n-1,b) \star m = a + b$$

$$\textsf{recdef}\ \$RecFn(f) := f \mapsto \forall n, a.$$
$$\{\$RecFn(a) \star res \mapsto \_\} \cdot (a,n)\, \{\exists v.\ \$RecFn(a) \star res \mapsto v * \$Rel(n,v)\}$$

$$\textsf{recdef}\ \$ListLibStrong(lookupL, addL, createL, disposeL) :=$$
$$lookupL \mapsto \ldots \star createL \mapsto \ldots \star disposeL \mapsto \ldots$$
$$\star\ addL \mapsto \forall al, key, value, \kappa.$$
$$\left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star\ \$Rel(key,value) \end{array} \right\} \cdot (al, key, value) \quad \left\{\ \$AssocListH(al; \{key\} \cup \kappa)\ \right\}$$

$$\textsf{recdef}\ \$ListLibWeak(lookupL, addL, createL, disposeL) :=$$
$$lookupL \mapsto \ldots \star createL \mapsto \ldots \star disposeL \mapsto \ldots$$
$$\star\ addL \mapsto \forall al, key, value.$$
$$\left\{ \begin{array}{l} \exists \kappa.\ \$AssocListH(al;\kappa) \\ \star\ \$Rel(key,value) \end{array} \right\} \cdot (al, key, value) \quad \left\{ \begin{array}{l} \exists \kappa. \\ \$AssocListH(al;\kappa) \end{array} \right\}$$

$$\textsf{recdef}\ \$S(a, f, al, lookupL, addL, createL, disposeL) \ :=\ \$RecFn(a) \circ DeepInv$$

where $DeepInv$ abbreviates

$$\exists \kappa. \left(\begin{array}{c} f \mapsto \forall n, a. \\ \left\{\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto \_\ \right\} \\ \cdot(a,n) \\ \left\{\ \exists v.\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto v \star \$Rel(n,v)\ \right\} \\ \star\ \$AssocListH(al;\kappa) \star \$ListLibWeak(lookupL, addL, createL, disposeL) \end{array} \right)$$

$$\textsf{recdef}\ \$AssocList(x;\tau) \quad := \quad x = 0 \star \tau = \emptyset$$
$$\vee\ \exists next, k, v, \tau'.\ x \mapsto k, v, next \star \$Rel(k,v) \star \$AssocList(next;\tau') \star \tau = \{k\} \cup \tau'$$

$$\textsf{recdef}\ \$AssocListH(x;\tau) := \exists y.\ x \mapsto y \star \$AssocList(y;\tau)$$

Fig. 4 User-defined predicates used to specify and verify our running example.

lists are manipulated via four library routines, pointers to which are passed in the arguments *lookupL*, *addL*, *createL*, *disposeL*. Argument f to procedure *mem* is a pointer to the code of the function being memoised; the memoiser must call this code when the required data is not found in the cache. The arguments *lookupL*, *addL*, *createL*, *disposeL*, al, f are fixed by partial application when the memoiser is first loaded onto the heap. This leaves a two-argument procedure: the first argument a is passed straight through to the function being memoised, and the second argument n is the input at which to apply the function.

The memoiser is designed to be placed into mutual recursion with *fib*, or similar code for computing other functions. During computations the *fib* code and the

proc $main()$
  pre : $res \mapsto \_$;
  post : emp;

proc $fib(a, n)$
  pre : $\$RecFn(a) \star res \mapsto \_$;
  post : $\exists v.\ \$RecFn(a) \star res \mapsto v \star \$Rel(n, v)$;

proc $mem(lookupL, addL, createL, disposeL, al, f, a, n)$
  pre : $\$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto \_$;
  post : $\exists m.\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto m \star \$Rel(n, m)$;

proc $useFib(lookupL, addL, createL, disposeL)$
  pre : $res \mapsto \_ \star \$ListLibWeak(lookupL, addL, createL, disposeL)$;
  post : emp;

proc $load\_list\_lib(lookupL, addL, createL, disposeL)$
  pre : $lookupL \mapsto \_ \star addL \mapsto \_ \star createL \mapsto \_ \star disposeL \mapsto \_$;
  post : $\$ListLibStrong(lookupL, addL, createL, disposeL)$;

Fig. 5 Procedure specifications for the memoiser example.

memoiser then invoke each other in a "zig-zag" mutual recursion. The "ensemble" of these two functions stored on the heap and able to invoke each other can be described by $\$S(a, f, al, lookupL, addL, createL, disposeL)$ which, as will be shown later by Lemma 7 (Section 6), is equivalent to:

$$\exists \kappa.\quad a \mapsto RecFnMem(\cdot) \star f \mapsto RecFnMem(\cdot) \\ \star\ \$AssocListH(al; \kappa)\ \star\ \$ListLibWeak(lookupL, addL, createL, disposeL)$$

where $RecFnMem(\cdot)$ is shorthand for

$$\forall a, n.\quad \frac{\left\{\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto \_\ \right\}}{\cdot(a, n)} \\ \left\{\ \exists v.\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto v \star \$Rel(n, v)\ \right\}$$

Intuitively *RecFnMem* describes code which computes a function as specified by $\$Rel$, provided the heap contains the "ensemble" of function and memoiser code as described above.

The main program. The *main* procedure first calls *load_list_lib* to load the association list library routines onto the heap. Then, *main* invokes *useFib* which loads the *fib* code and the memoiser, places them into mutual recursion, and finally uses this to compute the 31337th Fibonacci number.

In *useFib* we see the crucial role of the deep frame rule. We have specified (and therefore will verify) *fib* for the case where it is placed in recursion *only with itself*, using $\$RecFn$. Hence, if the deepframe annotation were not used in *useFib*, the symbolic heap after the statement $[f] := \text{proc } fib(\_, \_)$ would contain

$$f \mapsto \forall a, n.\, \{\$RecFn(a) \star res \mapsto \_\} \cdot (a, n)\, \{\exists v.\ \$RecFn(a) \star res \mapsto v \star \$Rel(n, v)\}$$

However the annotation deepframe *DeepInv* triggers the application of $- \otimes DeepInv$ to the above triple, resulting in $RecFnMem(\cdot)$. In this way, we have used the deep frame rule to *derive* another specification for the *fib* code, which describes how that code works in mutual recursion with a memoiser. We did not need to respecify or reprove *fib*.

The list library. The memoiser depends only on relatively weak properties of the association list library; a library with these properties is specified by the predicate $\$\mathit{ListLibWeak}$. But the list library is specified with a stronger specification $\$\mathit{ListLibStrong}$ so that it can also be used with other clients which need additional guarantees. Specifications for three of the routines are omitted in Figure 4, but with the remaining "add" routine one can see a difference. In order to compute the correct function, the memoiser does not care whether the $(\mathit{key}, \mathit{value})$ pair is actually added to the list or not, as long as whatever pairs are in the list afterwards are suitably related by $\$\mathit{Rel}$. But other clients of the list library will certainly care about this.

Our verification will go through because our rules for proving entailments are able to show

$$\begin{array}{rl} & \$\mathit{ListLibStrong}(\mathit{lookupL}, \mathit{addL}, \mathit{createL}, \mathit{disposeL}) \\ \Rightarrow & \$\mathit{ListLibWeak}(\mathit{lookupL}, \mathit{addL}, \mathit{createL}, \mathit{disposeL}) \end{array} \tag{2}$$

as we shall discuss in Section 6.5. Having such entailments proved automatically facilitates reasoning when one is "plugging together" different pieces of code.

4 A "core" Hoare logic for Higher-order Store

In this section we present and discuss the Hoare logic with respect to which our verification system is sound. We refer to those rules as the "core" rules. These rules are non-deterministic and not particularly suitable for automated proof search but they give a high-level logic for higher-order store. In Section 6 we will discuss judgements and rules for automated proof search and we prove that these rules are sound w.r.t. the "core" rules presented in this section. Note that the assertion language of this "high-level (core)" logic is richer than the assertion language of our verifier. For instance, there are no syntactic restrictions about quantifiers in assertions, conjunction and implication between assertions yields assertions, and *true* is an assertion. Similarly, arguments of procedures can be arbitrary expressions.

4.1 Rules for generating verification conditions (VCs)

Because programs use mutually recursive fixed procedures, the first rule to apply is a version of the well-knkown recursive procedure rule [25], the premise of which generates the verification conditions for the given program. It basically says that in order to prove a procedure correct wrt. its pre- and postcondition, we have to prove its body correct, assuming that recursive calls of the procedure already meet the specification.

Let $\mathcal{F}_1, \ldots, \mathcal{F}_n$ be the names of the concrete procedures declared in the program with bodies $\mathit{body}(\mathcal{F}_i)$ to meet specifications $\{\mathit{pre}(\mathcal{F}_i)\}\,\mathcal{F}_i(\mathit{params}(\mathcal{F}_i))\,\{\mathit{post}(\mathcal{F}_i)\}$ and let us abbreviate the context of these n triple specifications $\Gamma_{\mathcal{F}}$. Then the rule for the correctness of those specifications with respect to the procedure declarations is as follows:

RecursiveProcedures

$$\frac{\forall j \in \{1,\ldots,n\}.\ \left(\Pi; \Gamma_{\mathcal{A}}, \Gamma_{\mathcal{F}} \Vdash \left\{\mathit{pre}(\mathcal{F}_j)\right\}\, \mathit{body}(\mathcal{F}_j)\, \left\{\mathit{post}(\mathcal{F}_j)\right\}\right)}{\Pi; \Gamma_{\mathcal{A}} \Vdash \{\mathit{pre}(\mathcal{F}_i)\}\,\mathcal{F}_i(\mathit{params}(\mathcal{F}_i))\,\{\mathit{post}(\mathcal{F}_i)\}} \quad i \in \{1,\ldots,n\}$$

where the judgement $\Pi; \Gamma \Vdash \{P_i\}\, C_i\, \{Q_i\}$ expresses that Hoare triple $\{P_i\}\, C_i\, \{Q_i\}$ is derivable assuming the predicate definitions Π, containing equivalences $\mathrm{P}(\mathbf{x}) \Leftrightarrow Q$ which give meaning to the predicates, and procedure interface specifications Γ. In addition to the context $\Gamma_{\mathcal{F}}$ of triples for declared procedures we need a context $\Gamma_{\mathcal{A}}$ to account for any abstract procedures which of course do not have bodies. Variables appearing in these triples need to be treated as universally quantified around the triple (see semantics in Section 5).

4.2 Rules for proving VCs

If we have n non-abstract procedures then the rule (RecursiveProcedures) generates n verification conditions. Each VC has the form of a Hoare triple in context $\Pi; \Gamma \Vdash \{P\}\, C\, \{Q\}$, where C is a concrete command, which must be proved with respect to some information about user-defined predicates in Π and declared procedures in Γ. To prove the verification conditions we basically use a version of the logic in [40] with some changes that we shall shortly explain.

The rules in [40] enrich Separation Logic with rules for higher-order store. However, the programming language used in *loc. cit.* is more restricted in that it lacks the parameter passing, fixed procedures and mutable local variables available in the language we use here. In Figure 6, we list the syntax-driven rules. The full set of rules can be found in the Appendix A. We discuss their soundness in Section 5 and use them to prove soundness of the rather involved deterministic rules in Section 6 that implement the proof search.

Variable naming conventions Throughout the paper we will use the following variable names in rules to distinguish their kinds: name x usually denotes a program (and thus an integer) variable (or as list of variables $\mathbf{x}$); for parameter lists of functions usually $\mathbf{p}$ is in use which also denotes a list of integer variables. For list segments, s and t are used as additional integer variables. In the core logic we also use k typically as a variable representing (the code of) a stored procedure[4]. Names a, b, u, v, w and y (or as lists of variables $\mathbf{a}$ etc.) denote either integer *or* set variables.

Expression naming conventions Similarly for expressions that can denote either sets or integers the letter e (or $\mathbf{e}$ for lists of such expressions) is used (also E at times). The names e_A, e_V and e_S denote just address, integer value and set expressions, respectively. For set expressions we also use sometimes e_α or e_β (see the rules on list segments in Section 2.3.3). All expressions can appear with the usual extra decorations, so we may use e', e_1 or $\hat{e}$ or even e'_1 or $\hat{e_1}$ in cases where several expressions appear in one rule.

Substitution notation convention In reasoning rules we will write $e[x\backslash t]$ for the substitution of variable x in term e by a term t.

[4] In the implemented low level logic we never use such variables explicitly.

Explanation of Figure 6 The rules are mostly as found in other separation logic systems, providing variable assignment, heap allocation, deallocation and address dereferencing. The interesting rules here are those that utilise the higher-order store: storing code on the heap, and eval. Three notable rules are discussed below.

Rule (Call) is for reasoning about declared procedures. It is worth pointing out that the specification of the procedure is taken from the specification context and that parameter instantiation is done implicitly in the hypothesis. The hypothesis consists of an entailment between the declared specification of the procedure and its actual invocation. The universal quantification of free variables in procedure specifications becomes visible here. The entailment judgement is discussed below. The symbolic heap before the invocation P is likely to be larger than the procedure's footprint A which is dealt with as usual by the frame rule.

Rule (Eval) is for reasoning about stored procedures[5] and needs to deal with recursion through the store. The invoked procedure is stored on the heap at address e_A and is supposed to fulfil the triple $\{P\}\cdot(\mathbf{e_V})\,\{Q\}$ where P is the precondition of the eval command. Note that there are no universally quantified variables around this (nested) triple as parameter instantiation is done implicitly in the hypothesis like for the (Call) rule.

Rule (StoreProc) allows the loading of a fixed procedure from the context Γ into a single cell on the heap. The rule is complicated by the ability to use partial application, requiring some special handling of the $\forall$-quantified variables and the parameters. Essentially, the rule states that any argument provided in $\mathbf{t}$, say the i-th such argument t_i, that is *not* the underscore, replaces any occurrences of the i-th formal parameter in the code's specification. That parameter is then dropped from the parameter list, and also from the $\forall$-quantified variables such that it is bound outside the nested-triple. Any argument provided in $\mathbf{t}$, say the k-th such argument t_k, that *equals* the underscore implies that the k-th argument of the original function remains an argument in the resulting partially applied function. The addition of $\mathbf{y}$ to the $\forall$-variables ensures that no other variables used in the specification will be captured.

Rules for entailments between assertions. The judgement for entailments between assertions is

$$\Pi \Vdash P \Rightarrow Q$$

where Π is the set of predicate definitions which give meaning to predicates in assertions P and Q. We will often drop the Π annotation when it is irrelevant or obvious from the context.

Besides the usual basic properties of first order logic with equality, we use the rules as outlined in Appendix A.2 divided up in three categories. The first group describes the usual properties of Separation Logic connectives. The second is about the usual distribution of the tensor $\otimes$ used to frame on assertions "deeply" (see [40]), and the third is about nested triple entailments. Note that in our logic the first group also contains the axioms

$\star$-SplitPureLeft		$\star$-SplitPureRight	
$A \star \Phi \Rightarrow A$	if Φ is pure	$A \wedge (\Phi \star \mathit{true}) \Rightarrow A \star \Phi$	if Φ is pure

[5] We use a version of this rule that is different from the one in [40,14] as it reflects more closely the rule used for proof search.

$$\textsc{assign}\quad \{P\}\, x := e\, \{\exists x' \,.\, x = e[x\backslash x'] \star P[x\backslash x']\}$$

$$\textsc{lookup}\quad \{P \star e_A \mapsto e'\}\, x := [e_A]\, \{\exists x' \,.\, x = e'[x\backslash x'] \star (e_A \mapsto e')[x\backslash x'] \star P[x\backslash x']\}$$

$$\textsc{heap-assign}\quad \{e \mapsto \_\}\, [e_A] := e'\, \{e_A \mapsto e'\}$$

$$\textsc{heap2heap-assign}\quad \{e \mapsto \mathscr{C} \star e' \mapsto \_\}\, [e_A] := [e'_A]\, \{e_A \mapsto \mathscr{C} \star e'_A \mapsto \mathscr{C}\}$$

$$\textsc{new}\quad \{P\}\, x := \mathsf{new}\ e\, \{\exists x' \,.\, x \mapsto e[x\backslash x'] \star P[x\backslash x']\} \qquad \textsc{dispose}\quad \{e_A \mapsto \_\}\ \mathsf{dispose}\ e_A\ \{\mathsf{emp}\}$$

$$\textsc{Call}\quad \frac{\Pi \Vdash (\forall \mathbf{y}.\, \{A\}\, k(\mathbf{p})\, \{B\}) \Rightarrow \{P\}\, k(\mathbf{e_V})\, \{Q\}}{\Pi; \Sigma, \{A\}\, \mathcal{F}(\mathbf{p})\, \{B\} \Vdash \{P\}\, \mathsf{call}\ \mathcal{F}(\mathbf{e_V})\, \{Q\}}\ \mathbf{y} = fv(A, \mathbf{p}, B) \qquad k \text{ fresh}$$

$$\textsc{Eval}\quad \frac{P \Rightarrow e_A \mapsto \{P\} \cdot (\mathbf{e_V})\, \{Q\} \star \mathit{true}}{\{P\}\, \mathsf{eval}\ [e_A](\mathbf{e_V})\, \{Q\}} \qquad \textsc{skip}\quad \{P\}\, \mathsf{skip}\, \{P\}$$

$$\textsc{StoreProc}\quad \{A\}\, \mathcal{F}(\mathbf{p})\, \{B\} \Vdash \{e_A \mapsto \_\}\, [e_A] := \mathcal{F}(\mathbf{t})\, \{e_A \mapsto (\forall \mathbf{p}|_U, \mathbf{y}.\ \{A\} \cdot (\mathbf{p}|_U)\, \{B\})\, [\mathbf{p}|_{I\backslash U} \backslash \mathbf{t}|_{I\backslash U}]\}$$
$$\text{where } |\mathbf{t}| = |\mathbf{p}| \quad \text{and } t_i \text{ either value expression } a_i \text{ or } \_;\ \mathbf{y} = fv(A, B) - \mathbf{p};$$
$$\mathbf{p} = (p_i)_{i \in I};\quad U = \{i \in I \mid t_i = \_\}\quad \mathbf{p}|_X = (p_i)_{i \in I \cap X}$$

$$\textsc{scomp}\quad \frac{\Pi; \Gamma \Vdash \{P\}\, C_1\, \{R\} \quad \Pi; \Gamma \Vdash \{R\}\, C_2\, \{Q\}}{\Pi; \Gamma \Vdash \{P\}\, C_1; C_2\, \{Q\}}$$

$$\textsc{if}\quad \frac{\Pi; \Gamma \Vdash \{P \wedge e_V = e'_V\}\, C_1\, \{Q\} \quad \Pi; \Gamma \Vdash \{P \wedge e_V \neq e'_V\}\, C_2\, \{Q\}}{\Pi; \Gamma \Vdash \{P\}\, \mathsf{if}\ e_V = e'_V\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\, \{Q\}}$$

$$\textsc{while1}\quad \frac{\Pi; \Gamma \Vdash \{I \wedge e_V = e'_V\}\, C\, \{I\}}{\Pi; \Gamma \Vdash \{I\}\, \mathsf{while}\ e_V = e'_V\ \mathsf{do}\ C\, \{I \wedge e_V \neq e'_V\}} \qquad \textsc{while2}\quad \frac{\Pi; \Gamma \Vdash \{I \wedge e_V \neq e'_V\}\, C\, \{I\}}{\Pi; \Gamma \Vdash \{I\}\, \mathsf{while}\ e_V \neq e'_V\ \mathsf{do}\ C\, \{I \wedge e_V = e'_V\}}$$

Fig. 6 Syntax-driven rules for Hoare-triples

These axioms are needed as in the assertion language of the verifier we do not use $\wedge$ for adding pure assertions but rather $\star$.[6] The following axiom states that pure facts can be duplicated

$$\star\text{-}\textsc{IdemPure}\quad \Phi \Rightarrow \Phi \star \Phi \qquad \text{if } \Phi \text{ is pure}$$

and can be inferred from ($\star$-SplitPureRight) and the fact that $\Phi \Rightarrow \Phi \star \mathit{true}$ (which follows from the Separation Logic axioms).

The following rule will be used later to prove that invariant extension operator $\otimes$ distributes over recursive predicates (Lemma 7). It states that each recursive definition of a predicate has a unique solution. In order for this to hold we need to ensure that R in the rule below gives rise to a unique solution. This is not possible

[6] Semantically this means that a pure assertion Φ will only hold in the empty heap. Therefore Φ will not hold in the general sense (ie. for all heaps) but $\Phi \star \mathit{true}$ will.

within the logic of the tool but left as external proof-obligation

$$\textsc{RUnique}\ \frac{\forall \mathbf{y}.\ (R[\mathrm{X}\backslash P])(\mathbf{y}) \Leftrightarrow \mathrm{P}(\mathbf{y}) \qquad \forall \mathbf{y}.\ (R[\mathrm{X}\backslash Q])(\mathbf{y}) \Leftrightarrow \mathrm{Q}(\mathbf{y})}{\forall \mathbf{y}.\ P(\mathbf{y}) \Leftrightarrow Q(\mathbf{y})} R \text{ admits a unique solution}$$

where R is a formula denoting an assertion, with a free predicate variable X used with the appropriate arity and admits a unique solution. This is the case, for instance, if we R matches the pattern as described in [16] (so in particular it cannot be X itself).

Note that with the help of ∃-Introduction on the left hand side of the implication and (Consequence) one can derive (Skolem). Both are needed to eliminate existentially quantified variables. Both can be found in Figure 7.

For the sake of the proof search rules later, we now introduce two useful functions, *purify*(−) and *closure*(−), which transform spatial conjunctions and return spatial conjunctions. The function application $\mathit{purify}(A_1 \star \cdots \star A_n)$ returns the spatial conjunction of just those conjuncts A_i that are pure. The application $\mathit{closure}(\Phi)$ returns Φ conjoined with some extra pure facts which are already implicit in the spatial parts of Φ. For example if the spatial parts are $x \mapsto \_ \star y \mapsto 0$ the pure constraints $x \neq 0$, $y \neq 0$ and $x \neq y$ are added. These functions are designed to satisfy the following characteristic properties which we will add to the list of axioms:

$$\textsc{Purify}\quad \Phi \ \Leftrightarrow\ \mathit{purify}(\Phi) \star \Phi \qquad\qquad \textsc{Closure}\quad \Phi \ \Leftrightarrow\ \mathit{closure}(\Phi)$$

Due to the syntactic nature of the functions, the above axioms can be shown by simple induction on the structure of Φ using the laws of Separation Logic.

Rules for inductive list segment predicates. A limited form of inductive reasoning for list segments is available by means of the following two axioms:

$$\textsc{Join}\ \frac{\mathsf{recdef}\ \mathrm{L}(s,t;\alpha) := P \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k])}{\Pi, (\mathrm{L}(s,t;\alpha) \Leftrightarrow P) \Vdash \mathrm{L}(e_1, e, e_\alpha) \star \mathrm{L}(e, e_2; e_\beta) \Rightarrow \mathrm{L}(e_1, e_2; e_\alpha \cup e_\beta)}$$

$$\textsc{Split}\ \frac{\mathsf{recdef}\ \mathrm{L}(s,t;\alpha) := P \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k])}{\begin{array}{l}\Pi, (\mathrm{L}(s,t;\alpha) \Leftrightarrow P) \Vdash (e_1, \ldots, e_k) \in \hat{e}_\gamma \ \star\ \mathrm{L}(\hat{e}_1, \hat{e}_2, \hat{e}_\gamma) \Rightarrow \\ \qquad \exists s, n, \mathbf{v}, \alpha, \beta. \begin{pmatrix} \mathrm{L}(\hat{e}_1, s; \alpha) \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, \hat{e}_2, \beta) \\ \star\ \hat{e}_\gamma = \alpha \cup \{(e_1, \ldots, e_k)\} \cup \beta \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{pmatrix}\end{array}}$$

Rules for entailments between triples about procedures. Since our logic uses nested triples, entailment becomes more complicated and we need to prove entailments between such triples that talk about the behaviour of procedures. The rules can be found in Figure 7.

$$\textsc{ShallowFrameProcedures}\quad \frac{}{\{P\}\, e(\mathbf{e})\, \{Q\} \Rightarrow \{P \star R\}\, e(\mathbf{e})\, \{Q \star R\}} \qquad \textsc{ConsequenceProcedures}\quad \frac{P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P'\}\, k(\mathbf{e_V})\, \{Q'\} \Rightarrow \{P\}\, k(\mathbf{e_V})\, \{Q\}}$$

$$\textsc{Disjunction}\quad \frac{A \Rightarrow \{\Psi_1\}\, k(\mathbf{e_V})\, \{Q\} \qquad A \Rightarrow \{\Psi_2\}\, k(\mathbf{e_V})\, \{Q\}}{A \Rightarrow \{\Psi_1 \vee \Psi_2\}\, k(\mathbf{e_V})\, \{Q\}}$$

$$\textsc{Skolem}\quad \frac{}{\{\Phi[x \backslash x_0]\}\, k(\mathbf{e_V})\, \{Q\} \Rightarrow \{\exists x.\Phi\}\, k(\mathbf{e_V})\, \{Q\}} \quad x_0 \text{ fresh}$$

Fig. 7 Rules for entailment between behavioural specifications

5 Soundness of the logic above

To show soundness of the rules above we adopt and adapt a model and proofs that have been presented in [7].

We will recapitulate the techniques of this "hybrid" model that uses operational semantics and step-indexing but also denotational semantics to construct the type of Kripke-worlds used for modelling the deep frame rule. There might be other ways to construct models for our language but the model presented has various advantages, including notably that it can be easily extended to include the *anti-frame rule* [34, 42] for future extensions of our logic. The rules in Section 6 that do the automatic proof search are then shown sound relative to the rules above. The main differences between the rules presented here and the ones in [40] are the use of recursively definable procedures with (integer) parameters and local variables[7] and the use of those procedures as code expressions (instead of quoted code) which allows for partial application at the point of heap update. The main feature of the presented model is that assertions are step-indexed predicates on heaps further indexed by worlds representing the (specification) invariants that can be framed on to allow for deep framing.

5.1 Operational semantics of programming language

We define a small-step operational semantics as defined in Figure 8. A configuration (C, s, h) of the semantics consists of an open command C, a stack of variable environments $s \in \mathit{Stack}$, and a heap $h \in \mathit{Heap}$. For stacks of environments $s \cdot \eta$ refers to a stack with topmost environment η and $\emptyset$ denotes the empty stack. All components of a configuration may change during execution. In order to deal with procedure calls there is also a constant environment for procedure declarations γ that is not included in the configuration as it is fixed and will not change. The reduction relation however carries γ as superscript to indicate the dependency.

We next define the semantic domains used in configurations, that is heaps, variable environments and procedure environments: First, the domain of heaps is

[7] In our language the syntax is slightly different from *loc.cit.* as we declare all local variables at the beginning of procedures instead of using the let... in... syntax and the local variables can be updated.

defined as

$$Heap = \mathbb{N}_{>0} \rightharpoonup_{\mathsf{fin}} \mathbb{Z}$$

ie. partial maps from strictly positive[8] natural numbers (the addresses) to integers that have finite domain. Therefore, heaps store only integers which has the advantage that they are "flat" and so do not need to be defined as recursive types containing objects of higher-order, thus raising problems w.r.t. admissibility of predicates on heaps. Despite being technically "flat" they still are "higher-order" in spirit as we store procedures in them via a simple encoding $\lceil \_ \rceil$. As usual, the empty heap is abbreviated e and there is a partial binary operation $h_1 \cdot h_2$ that adjoins two heaps h_1 and h_2 if $h_1 \# h_2$ which abbreviates that h_1 and h_2 have disjoint domains, ie. $\mathsf{dom}(h)_1 \cap \mathsf{dom}(h)_2 = \emptyset$.

Variable environments are defined as *total* maps from variable names to integer values.

$$Env = Var \to \mathbb{Z}$$

Variable names include program variable names as well as auxiliary variable names used to express specifications. One can only quantify over auxiliary variables. There is an update operation

$$\eta[x \mapsto e](y) = \begin{cases} e & \text{if } x = y \\ \eta(y) & \text{otherwise} \end{cases}$$

Using total maps allows us to ignore variable declarations in the semantics. This means our failure avoiding semantics of Hoare triples will not be able to guarantee that programs don't use undeclared variables. The front-end of our verifier will only accept program and specifications where variables are correctly declared.

Finally, a procedure environment γ maps procedure names $\mathcal{F}$ to declarations $\mathsf{proc}\,(\mathbf{x})\{\mathsf{locals}\ \mathbf{y}; C\}$ in syntactic form.

The operational semantics is a relation that relates two configurations and a program. A configuration consists of a stack of environments and a heap. A final configuration is a normal configuration or a special *aborting* configuration abort. This special (terminal) configuration is reached if a memory fault or any kind of runtime error occurs, for instance if for a procedure call the actual argument number does not match the arity of the procedure. The rules for the non aborting cases in the operational semantics are given in Figure 8; the aborting cases can be found in Figure 9.

In addition to the usual variable environment η that maps variables to integers, we need a stack of such environments as we use a language with procedure blocks that have local variables. The procedure environment γ that maps procedure names to their definitions is used for interpreting call statements by looking up the corresponding procedure in the environment. Again, if the arguments do not match the procedure's arity, we abort. For storing procedures we need to perform partial application. Thus we define the operation

$$\mathit{papply}\ (\mathsf{proc}\,(\mathbf{x})\{\mathsf{locals}\,\mathbf{y}; C\})\ (t_1, \ldots, t_n) = \mathsf{proc}(\mathbf{x}|_U)\{\mathsf{locals}\,\mathbf{y}; C[\mathbf{x}|_{I\setminus U} \backslash \mathbf{t}_{I\setminus U}]\}$$

where $\mathbf{x} = (x_i)_{i\in I}$, $U = \{i \in I \mid t_i = \_\}$ and $\mathbf{x}|_X = (x_i)_{i\in I\cap X}$. This operation substitutes those actual parameters t_i of $\mathbf{t}$ that are not $\_$ for the matching formal

[8] The address 0 denotes the nil pointer.

$$
\begin{array}{rcl}
(x := e_V,\ s\cdot\eta,\ h) & \leadsto & (\mathsf{skip},\ s\cdot\eta[x \mapsto [\![e_V]\!]_\eta],\ h) \\
(x := \texttt{[}e_A\texttt{]},\ s\cdot\eta,\ h) \text{ and } [\![e_A]\!]_\eta \in \mathsf{dom}(h) & \leadsto & (\mathsf{skip},\ s\cdot\eta[x \mapsto h([\![e_A]\!]_\eta)],\ h) \\
(\texttt{[}e_A\texttt{]} := e_V,\ s\cdot\eta,\ h) & \leadsto & (\mathsf{skip},\ s\cdot\eta,\ h[[\![e_A]\!]_\eta \mapsto [\![e_V]\!]_\eta]) \\
 & & \text{if } [\![e_A]\!]_\eta \in \mathsf{dom}(h) \\
(\texttt{[}e_1\texttt{]} := \texttt{[}e_2\texttt{]},\ s\cdot\eta,\ h) & \leadsto & (\mathsf{skip},\ s\cdot\eta,\ h[[\![e_1]\!]_\eta \mapsto h([\![e_2]\!]_\eta)]) \\
 & & \text{if } [\![e_1]\!]_\eta \in \mathsf{dom}(h) \text{ and } [\![e_2]\!]_\eta \in \mathsf{dom}(h) \\
(x := \mathsf{new}\, e_1, \ldots, e_n,\ s\cdot\eta,\ h) & \leadsto & (\mathsf{skip},\ s\cdot\eta,\ h\cdot[a \mapsto [\![e_1]\!]_\eta]\cdot[a+1 \mapsto [\![e_2]\!]_\eta] \\
 & & \qquad\qquad \cdot \ldots \cdot [a+n{-}1 \mapsto [\![e_n]\!]_\eta]) \\
 & & \text{if } x \in \mathsf{dom}(\eta) \text{ and } a \notin \mathsf{dom}(h), \ldots a+n{-}1 \notin \mathsf{dom}(h) \\
(\mathsf{dispose}\, e,\ s\cdot\eta,\ h) & \leadsto & (\mathsf{skip},\ s\cdot\eta,\ h') \\
 & & \text{if } [\![e]\!]_\eta = n \text{ and } h = h' * [n \mapsto v] \\
(\mathsf{call}\ \mathcal{F}(e_1, \ldots, e_n),\ s\cdot\eta,\ h) & \leadsto^\gamma & (C; \mathsf{return},\ s\cdot\eta\cdot\eta[x_1 \mapsto [\![e_1]\!]_\eta, \ldots, x_n \mapsto [\![e_n]\!]_\eta, \\
 & & \qquad\qquad y_1 \mapsto 0, \ldots, y_m \mapsto 0],\ h) \\
 & & \text{if } \mathcal{F} \in \mathsf{dom}(\gamma) \text{ and} \\
 & & \quad \gamma(\mathcal{F}) = \mathsf{proc}\,(x_1, \ldots x_n)\{\mathsf{locals}\, y_1, \ldots, y_m; C\} \\
(\mathsf{return},\ s\cdot\eta,\ ,h) & \leadsto & (\mathsf{skip},\ s,\ h) \\
(\mathsf{eval}\ \texttt{[}e_A\texttt{]}(e_1, \ldots, e_n),\ s\cdot\eta,\ h) & \leadsto & (C; \mathsf{return},\ s\cdot\eta\cdot\eta[x_1 \mapsto [\![e_1]\!]_\eta, \ldots, x_n \mapsto [\![e_n]\!]_\eta, \\
 & & \qquad\qquad y_1 \mapsto 0, \ldots, y_m \mapsto 0],\ h) \\
 & & \text{if } [\![e_A]\!]_\eta \in \mathsf{dom}(h) \text{ and} \\
 & & \quad \lceil [\![e_A]\!]_\eta \rceil^{-1} = \mathsf{proc}\,(x_1, \ldots x_n)\{\mathsf{locals}\, y_1, \ldots, y_m; C\} \\
(\texttt{[}e_A\texttt{]} := \mathsf{proc}\ \mathcal{F}(t_1, \ldots, t_n),\ s\cdot\eta,\ h) & \leadsto^\gamma & (\mathsf{skip},\ s\cdot\eta,\ h[[\![e_A]\!]_\eta \mapsto \lceil \mathit{papply}\ \gamma(\mathcal{F})\ (u_1, \ldots, u_n)\rceil]) \\
 & & \quad \text{where } u_i = \begin{cases} \_ & \text{if } \ t_i = \_ \\ \eta(x_i) & \text{if } \ t_i = x_i \end{cases} \\
 & & \text{if } [\![e_A]\!]_\eta \in \mathsf{dom}(h),\ \mathcal{F} \in \mathsf{dom}(\gamma), \text{ and arity } (\gamma(\mathcal{F})) = n \\
(C_1; C_2,\ s,\ \ h) & \leadsto & (C_1'; C_2,\ s',\ h') \\
 & & \text{if } (C_1,\ s,\ h) \leadsto (C_1',\ s',\ h') \\
(\mathsf{skip}; C_2,\ s,\ h) & \leadsto & (C_2,\ s,\ h) \\
(\mathsf{if}\ e_1 \bowtie e_2\ \mathsf{then}\, C_1\ \mathsf{else}\, C_2,\ s\cdot\eta,\ h) & \leadsto & (C_1,\ s\cdot\eta,\ h) \\
 & & \text{if } [\![e_1]\!]_\eta \bowtie [\![e_2]\!]_\eta \\
(\mathsf{if}\ e_1 \bowtie e_2\ \mathsf{then}\, C_1\ \mathsf{else}\, C_2,\ s\cdot\eta,\ h) & \leadsto & (C_2,\ s\cdot\eta,\ h) \\
 & & \text{if } [\![e_1]\!]_\eta \not\bowtie [\![e_2]\!]_\eta \\
(\mathsf{while}\, e_1 \bowtie e_2\ \mathsf{do}\, C,\ s\cdot\eta,\ h) & \leadsto & (C; \mathsf{while}\, e_1 = e_2\ \mathsf{do}\, C,\ s\cdot\eta,\ h) \\
 & & \text{if } [\![e_1]\!]_\eta \bowtie [\![e_2]\!]_\eta \\
(\mathsf{while}\, e_1 \bowtie e_2\ \mathsf{do}\, C,\ s\cdot\eta,\ h) & \leadsto & (\mathsf{skip},\ s\cdot\eta,\ h) \\
 & & \text{if } [\![e_1]\!]_\eta \not\bowtie [\![e_2]\!]_\eta
\end{array}
$$

Fig. 8 Operational semantics of our programming language. Here $\bowtie \in \{=, \neq, <, \leq\}$.

parameters x_i. For those $t_j = \_$ the formal parameter x_j is left alone such that the resulting procedure has arity $|U|$. Note that because we have outlawed assignments to formal parameters inside a procedure body, this substitution produces only well-formed statements and can*not* create "statements" like $3 := 4$.

Let us define Safe_n^γ to be the set of configurations in the operational semantics that are "safe" for n reduction steps, meaning the set of configurations that do not reduce to abort in n or fewer steps. Defining further $\leadsto_k^\gamma$ to be the restriction of the operational semantics with fixed procedure environment γ to k-steps, we can explicitly write

$$
\begin{array}{rcl}
(x := [e_V],\ s \cdot \eta,\ ,h) & \rightsquigarrow & \mathsf{abort} \\
& & \text{if } [\![e_V]\!]_\eta \notin \mathsf{dom}(h) \\
([e_A] := e_V,\ s \cdot \eta,\ ,h) & \rightsquigarrow & \mathsf{abort} \\
& & \text{if } [\![e_A]\!]_\eta \notin \mathsf{dom}(h) \\
([e_1] := [e_2],\ s \cdot \eta,\ h) & \rightsquigarrow & \mathsf{abort} \\
& & \text{if } [\![e_1]\!]_\eta \notin \mathsf{dom}(h) \text{ or } [\![e_2]\!]_\eta \notin \mathsf{dom}(h) \\
(\mathsf{dispose}\, e,\ s \cdot \eta,\ h) & \rightsquigarrow & \mathsf{abort} \\
& & \text{if } [\![e]\!]_\eta \notin \mathsf{dom}(h) \\
(\mathsf{call}\ \mathcal{F}(e_1, \ldots, e_n),\ s \cdot \eta,\ h) & \rightsquigarrow^\gamma & \mathsf{abort} \\
& & \text{if } \mathcal{F} \notin \mathsf{dom}(\gamma) \text{ or arity } \gamma(\mathcal{F}) \neq n \\
(\mathsf{return},\ \emptyset,\ h) & \rightsquigarrow & \mathsf{abort} \\
(\mathsf{eval}\ [e_A](e_1, \ldots, e_n),\ s \cdot \eta,\ h) & \rightsquigarrow & \mathsf{abort} \\
& & \text{if } [\![e_A]\!]_\eta \notin \mathsf{dom}(h) \text{ or} \\
& & \quad \lceil [\![e]\!]_\eta \rceil^{-1} \neq \mathsf{proc}\,(x_1, \ldots x_n)\{\mathsf{locals}\, y_1, \ldots, y_m; C\} \\
([e_A] := \mathsf{proc}\ \mathcal{F}(t_1, \ldots, t_n),\ s \cdot \eta,\ h) & \rightsquigarrow^\gamma & \mathsf{abort} \\
& & \text{if } [\![e_A]\!]_\eta \notin \mathsf{dom}(h) \text{ or } \mathcal{F} \notin \mathsf{dom}(\gamma) \text{ or arity } \gamma(\mathcal{F}) \neq n \\
\\
(C_1; C_2,\ s,\ h) & \rightsquigarrow & \mathsf{abort} \\
& & \text{if } (C_1,\ s,\ h) \rightsquigarrow \mathsf{abort}
\end{array}
$$

Fig. 9 Abort cases of the operational semantics of our programming language

$$\mathrm{Safe}_n^\gamma = \{\Delta \in \mathrm{Config} \mid \neg \exists k \leq n.\, \Delta \rightsquigarrow_k^\gamma \mathsf{abort}\}$$

5.2 Semantics of assertions including triples

Again, we follow the ideas of [7] and adapt and extend them according to our language. The first main idea here is to use step-indexed predicates. So let $\mathit{UPred}(H)$ (for any H) be the set of subsets of $\mathbb{N} \times H$ that are downwards closed in the index part (first component):

$$\{p \subseteq \mathbb{N} \times H \mid \forall (k,h) \in p.\, \forall j \leq k.\, (j,h) \in p\}.$$

For H we pick the ingredients we need to interpret assertions so

$$H = \mathit{Env} \times \mathit{SetEnv} \times \mathit{Heap}$$

where *SetEnv* is the domain of environments that map auxiliary set variables (α) to sets of "element values". Element values can be integers, but also tuples of element values and sets of element values, ie.

$$\mathit{Set} = \mathbb{P}(\mathit{Elem}) \qquad \mathit{Elem} = \mathbb{Z} + \mathit{Tuple} + \mathit{Set} \qquad \mathit{Tuple} = \sum_n (\mathit{Elem})^n$$

To equip $\mathit{UPred}(H)$ with a distance function we first define a restriction operator $p_{[n]}$ for any $p \in \mathit{UPred}(H)$ as follows:

$$p_{[n]} := \{(k,v) \in p \mid k < n\}$$

So by definition for all predicates p and q we obtain $p_{[0]} = \emptyset = q_{[0]}$. Now we can define a distance function for $\mathit{UPred}(H)$ as follows

$$\delta(p,q) := \inf\{2^{-n} \mid p_{[n]} = q_{[n]}\}$$

which by the above observation is bounded by 1. That it is a distance map with the right properties can be shown easily. For the ultrametric version of the triangular inequality one needs the property

$$(p_{[n]})_{[m]} = p_{[\min(n,m)]}$$

Let $\mathsf{CBUlt_{ne}}$ denote the category of complete 1-bounded non-empty ultra metric spaces which is used to interpret predicates of our logic.

For two elements e and $e' \in A \in \mathsf{CBUlt_{ne}}$ we write $e \overset{n}{=} e'$ for $\delta(e,e') = 2^{-n}$.

As a consequence $\mathit{UPred}(H) \in \mathsf{CBUlt_{ne}}$, i.e. $\mathit{UPred}(H)$ is a complete, 1-bounded ultrametric space; for a proof of this see [7]. Using further results cited in *loc.cit.* [Theorem 2.1] we obtain a unique $W \in \mathsf{CBUlt_{ne}}$ satisfying

$$W \cong \tfrac{1}{2}(W \to \mathit{UPred}(H))\,. \tag{3}$$

We define $\mathit{Pred} = \frac{1}{2}(W \to \mathit{UPred}(H))$ so we can denote the isomorphism from (3)

$$\iota : \mathit{Pred} \to W \tag{4}$$

and we will refer to this ι frequently in this chapter. Assertions are to be modelled as elements of *Pred*. The "shrinking factor" $\frac{1}{2}$ for the metric automatically turns the function space of non-expansive maps between $\mathsf{CBUlt_{ne}}$s into a space of only contractive maps, for which then a solution of the recursive equation (3) exists up to isomorphism [38].

The set *Pred* is ordered pointwise:

$$p \leq q \iff \forall w \in W.\, p(w) \subseteq q(w)$$

As explained in the extended version of [7] we can show that *Pred* is a complete BI-algebra (in the sense of [35]):

Lemma 1 (*Pred* is a complete BI-algebra) *We can define all BI-operations:*

$$\begin{aligned}
\mathrm{emp}(w) &= \{(n,\eta,\sigma,\mathsf{e}) \mid n \in \mathbb{N}, \eta \in \mathit{Env}, \sigma \in \mathit{SetEnv}\}\\
(p \star q)(w) &= \{(n,\eta,\sigma,h) \mid \exists h_1, h_2.\, h = h_1 \cdot h_2\\
&\qquad \wedge (n,\eta,\sigma,h_1) \in p(w) \wedge (n,\eta,\sigma,h_2) \in q(w)\}\\
(p \mathrel{-\!\!*} q)(w) &= \{(n,\eta,\sigma,h) \mid \forall m \leq n.\\
&\qquad ((m,\eta,\sigma,h') \in p(w) \wedge h \# h') \implies (m,\eta,h \cdot h') \in q(w)\}
\end{aligned}$$

Proof Note that the quantification in the definition of $\mathrel{-\!\!*}$ is necessary to enforce a downward-closed predicate. The proofs are straightforward.

The fact that *Pred* is a complete BI algebra immediately gives us a sound interpretation of most of the assertions in the logic of [6], but to interpret recursive predicates we also need to know that the operations are non-expansive:

Lemma 2 *The BI-algebra operations on Pred given by the previous lemma are non-expansive:*

$$\cdot, -\!\!*, \to, \wedge, \vee : \mathit{Pred} \times \mathit{Pred} \to \mathit{Pred}$$
$$\textstyle\bigvee_I, \bigwedge_I : (I \to \mathit{Pred}) \to \mathit{Pred}.$$

(In the last two operations, the indexing set I is given the discrete metric.)

Proof For instance, $\star : \mathit{UPred}(H) \times \mathit{UPred}(H) \to \mathit{UPred}(H)$: It suffices to show that if $p \stackrel{n}{=} p'$ and $q \stackrel{n}{=} q'$, then also $(p \star q) \stackrel{n}{=} (p' \star q')$. The latter is equivalent to $\forall m < n.\, (m, \eta, \sigma, h) \in p \star q \iff (m, \eta, \sigma, h) \in p' \star q'$ following easily from the assumption and the definition of $\star$.

In the following, let us write $\top$ for the top element of *Pred* which is by definition $\lambda w \in W.\, \mathbb{N} \times H = \lambda w \in W.\, \mathbb{N} \times \mathit{Env} \times \mathit{SetEnv} \times \mathit{Heap}$.

5.2.1 Interpretation of invariant extension

To interpret invariant-extension assertions $P \otimes Q$, we need an operator $\otimes$ on the set of semantic predicates *Pred*. Working with metric-spaces, such a (unique) operator can be defined using Banach's fixed point theorem:

Lemma 3 *There exists a unique function $\otimes : \mathit{Pred} \times W \to \mathit{Pred}$ in the (non-expansive) function space of $\mathsf{CBUlt}_{\mathsf{ne}}$ satisfying*

$$p \otimes w = \lambda w'.\, p(w \circ w')$$

where $\circ : W \times W \to W$ is given by

$$w_1 \circ w_2 = \iota((\iota^{-1}(w_1) \otimes w_2) \cdot \iota^{-1}(w_2))\,.$$

Proof Both operations $\otimes$ and $\circ$ are mutually recursively defined by the above and their fixpoints exist by Banach's fixpoint theorem (see [41]).

5.2.2 Interpretation of triples

One notable difference w.r.t the interpretation of triples in [7,40] is that nested triples (behavioural specs B) now have additional parameters (possibly zero). Therefore, semantic triples now need to work on procedures instead of commands. We define a semantic interpretation of such Hoare triples next. Recall that we write $\rightsquigarrow_k^\gamma$ for the k-step reduction relation of the operational semantics.

Definition 1 Let $p, q \in \mathit{Pred}$, $w \in W$, $\eta \in \mathit{Env}$, $\sigma \in \mathit{SetEnv}$, let C be a program statement and let γ be a procedure environment. We define that $w, \eta, \sigma \models_n^\gamma (p, C, q)$ holds iff the following holds: for all $r \in \mathit{UPred}(H)$, all $m < n$, all heaps h, all stacks s, if $(m, \eta, \sigma, h) \in p(w) \star \iota^{-1}(w)(\mathrm{emp}) \star r$, then:

1. $(C, s \cdot \eta, h) \in \mathrm{Safe}_m^\gamma$.
2. For all $k \leq m$ and all $h' \in \mathit{Heap}$, $\eta' \in \mathit{Env}$, if $(C, s \cdot \eta, h) \rightsquigarrow_k^\gamma (\mathsf{skip}, s \cdot \eta', h')$, then $(m - k, \eta', \sigma, h') \in q(w) \star \iota^{-1}(w)(\mathrm{emp}) \star r$.

We write $n \models^{\gamma}_{\pi} (P, C, Q)$ iff for all $w \in W$ and for all set environments σ and integer environments η it holds that $w, \eta, \sigma \models^{\gamma}_{n} (\llbracket P \rrbracket_{\pi}, C, \llbracket Q \rrbracket_{\pi})$. Accordingly we write $\models^{\gamma}_{\pi} (P, C, Q)$ for $\forall n \in \mathbb{N}.\, n \models^{\gamma}_{\pi} (P, C, Q)$.

This definition is similar to the one in [40] with its use of the invariant w and the baking-in of the first order frame rule, i.e., the quantification over r. The difference is that the meaning is now relative to the operational semantics (rather than denotational) using the fixed procedure declarations in γ, and that we use step indexing to measure to what extent pre- and postconditions should hold. This idea has also appeared in the (unpublished) appendix of [7]. Note how the definition of $\models^{\gamma}_{\pi} (p, C, q)$ universally quantifies all free variables on the top level.

The intention is, of course, that a Hoare-triple assertion is interpreted using the above semantic construct (and this will be seen in Figure 11). The following lemma is crucial to achieve that this definition yields a non-expansive map:

Lemma 4 *If $w \overset{k}{=} w'$ and $w, \eta, \sigma \models_{n} (p, C, q)$, then $w', \eta, \sigma \models_{\min(n,k-1)} (p, C, q)$.*

Proof Straightforward verification, using Definition 1, the fact that if $w \overset{k}{=} w'$ then $\iota^{-1}(w)(\mathrm{emp}) \overset{k-1}{=} \iota^{-1}(w')(\mathrm{emp})$, the fact that the separating conjunction $\star$ is non-expansive on $\mathit{UPred}(H)$ and the definition of the distance map (Lemma 2). It is worth noting that to show this lemma the index m of tuples in $w, \eta, \sigma \models_{n} (p, C, q)$ as given in Definition 1 above must indeed be strictly smaller than n[9].

5.2.3 Interpretation of our assertion language

The interpretation of an assertion P is now defined to be an element $\llbracket P \rrbracket_{\pi}$ in Pred, where π is the environment for predicate definitions mapping predicate names (like P) to predicates in $\mathit{UPred}(H)$ of some finite arity. So π maps predicate names to functions mapping argument integer and set variables to a predicate in $\mathit{UPred}(H)$. How the declarations are interpreted will be discussed further below.

The definition uses the complete BI-algebra structure on Pred given earlier to interpret the standard logical connectives, e.g.,

$$\llbracket P \star Q \rrbracket_{\pi}\, w = \llbracket P \rrbracket_{\pi}\, w \star \llbracket Q \rrbracket_{\pi}\, w.$$

Invariant extension is interpreted as follows:

$$\llbracket P \otimes Q \rrbracket_{\pi}\, w = \Big(\llbracket P \rrbracket_{\pi} \otimes \iota(\llbracket Q \rrbracket_{\eta}) \Big)\, w$$

It is worth mentioning that we interpret the logic of the previous Section 4 which uses a superset of the assertion language defined in Figure 3 earlier. In particular, the "extended" logic includes general universal quantification, conjunction and implications between assertions and (stand-alone) triples as assertions. This allows one to express implications between triples which will be used later to verify the proof search rules for entailment (see Section 6).

The concrete interpretation of the logical connectives including implication and triples can be found in Figure 11. Note that we need to extend the interpretation

[9] This also guarantees that certain recursive definitions of predicates are contractive and thus admit a fixpoint. The existence of recursively defined predicates is, however, not discussed here but in [16].

$$
\begin{aligned}
[\![e_V]\!]_{\eta,\sigma} &= [\![e_V]\!]_\eta \qquad \text{as usual} \\
[\![(e_E^1,\ldots,e_E^n)]\!]_{\eta,\sigma} &= ([\![e_E]\!]^1_{\eta,\sigma},\ldots,[\![e_E^n]\!]_{\eta,\sigma}) \\
[\![\alpha]\!]_{\eta,\sigma} &= \sigma(\alpha) \\
[\![e_S^1 \cup e_S^2]\!]_{\eta,\sigma} &= [\![e_S^1]\!]_{\eta,\sigma} \cup [\![e_S^2]\!]_{\eta,\sigma} \\
[\![\{e_E\}]\!]_{\eta,\sigma} &= \{[\![e_E]\!]_\eta\} \\
[\![\emptyset]\!]_{\eta,\sigma} &= \{\} \\
[\![\mathrm{proj}_n(e_S)]\!]_{\eta,\sigma} &= \{\pi_n(v) \mid v \in [\![e_S]\!]_{\eta,\sigma}\}
\end{aligned}
$$

Fig. 10 Interpretation of expressions.

of expressions e_V from the operational semantics to assertion expressions e_E (including set expressions e_S) which requires the interpretation function to have an extra argument σ for set variables. The details can be found in Figure 10. Note that the abstract syntax already distinguishes between sets (e_S) and integers (e_V) so we do not need to do any type checking. Since we use a flat store the storable procedures (as expressions) are integers and live in e_V. They could, in principle, be used for arithmetic computations but since nothing about the encoding of procedures is axiomatised in the logic one would not be able to prove anything for such unintended uses.

Figure 11 does not contain a clause for $e \mapsto \_$ as this can be viewed as an abbreviation for $\exists v.\, e \mapsto v$. Similarly Figure 11 contains no clause for $e \mapsto \forall \mathbf{x}\,\{P\} \cdot (\mathbf{y})\,\{Q\}$ as this can be viewed as an abbreviation for $\exists c.(e \mapsto c \wedge \forall \mathbf{x}\,\{P\}\,c(\mathbf{y})\,\{Q\})$ for a fresh c.

Note that we could also interpret the spatial implication operator in this model as $[\![P \mathrel{-\!\!*} Q]\!]_\pi\, w = [\![P]\!]_\pi\, w \mathrel{-\!\!*} [\![Q]\!]_\pi\, w$. But we don't need a semantics for spatial implication as our tool does not support it yet.

Lemma 5 *The interpretation of assertions given in Figure 11 is well defined, ie. all denotations are non-expansive maps of type $W \to \mathit{UPred}(H)$.*

Proof Straightforward, the most complicated case is for nested triples but follows then easily from Lemma 4 using the fact that $[\![\{P\}e(t_1,\ldots,t_n)\{Q\}]\!]_\pi\, w \overset{k}{=} [\![\{P\}e(t_1,\ldots,t_n)\{Q\}]\!]_\pi\, w'$ if, and only if, for all $n < k$, for all η, σ and for γ' as defined in the Figure it holds that $w, \eta, \sigma \models^{\gamma'}_n ([\![P]\!]_\pi, \mathsf{call}\ \mathcal{N}(\mathbf{t}), [\![Q]\!]_\pi) \Leftrightarrow w', \eta, \sigma \models^{\gamma'}_n ([\![P]\!]_\pi, \mathsf{call}\ \mathcal{N}(\mathbf{t}), [\![Q]\!]_\pi)$.

5.2.4 The environment for predicate definitions

For interpreting predicate tests like $\mathrm{P}(\mathbf{e_v}; \mathbf{e_s})$ we use a predicate environment that maps each predicate identifier P to a specific semantic predicate of type $\mathbb{Z}^{\mathit{arity}_i(\mathrm{P})} \times \mathit{Set}^{\mathit{arity}_s(\mathrm{P})} \to \mathit{Pred}$ and assume that predicates are always used with the right arity, otherwise the predicate test is equivalent to false (of course, our tool uses a syntax checker that would reject assertions that use a wrong number of arguments).

As in [40, 7], recursively defined predicates are interpreted via Banach's fixed point theorem:

$$
\begin{aligned}
[\![\mathit{true}]\!]_\pi &= \top \\
[\![\mathit{false}]\!]_\pi\, w &= \emptyset \\
[\![P \vee Q]\!]_\pi\, w &= [\![P]\!]_\pi\, w \cup [\![Q]\!]_\pi\, w \\
[\![P \wedge Q]\!]_\pi\, w &= [\![P]\!]_\pi\, w \cap [\![Q]\!]_\pi\, w \\
(n,\eta,\sigma,h) \in [\![\exists x.\, P]\!]_\pi\, w &\text{ iff } \exists v \in \mathbb{Z}.\, (n, \eta[x \mapsto v], \sigma, h) \in [\![P]\!]_\pi\, w \\
(n,\eta,\sigma,h) \in [\![\forall x.\, P]\!]_\pi\, w &\text{ iff } \forall v \in \mathbb{Z}.\, (n, \eta[x \mapsto v], \sigma, h) \in [\![P]\!]_\pi\, w \\
(n,\eta,\sigma,h) \in [\![\forall \alpha.\, P]\!]_\pi\, w &\text{ iff } \forall v \in \mathit{Set}.\, (n, \eta, \sigma[\alpha \mapsto v], h) \in [\![P]\!]_\pi\, w \\
(n,\eta,\sigma,h) \in [\![P \Rightarrow Q]\!]_\pi\, w &\text{ iff } \forall m \leq n. \\
&\qquad (m,\eta,\sigma,h) \in [\![P]\!]_\pi\, w \text{ implies } (m,\eta,\sigma,h) \in [\![Q]\!]_\pi\, w \\
(\_,\eta,\_,h) \in [\![e^1_V = e^2_V]\!]_\pi\, w &\text{ iff } [\![e^1_V]\!]_\eta = [\![e^2_V]\!]_\eta \text{ and } h = \mathsf{e} \\
(\_,\eta,\_,h) \in [\![e^1_V \neq e^2_V]\!]_\pi\, w &\text{ iff } [\![e^1_V]\!]_\eta \neq [\![e^2_V]\!]_\eta \text{ and } h = \mathsf{e} \\
(\_,\eta,\sigma,h) \in [\![e^1_S = e^2_S]\!]_\pi\, w &\text{ iff } [\![e^1_S]\!]_{\eta,\sigma} = [\![e^2_S]\!]_{\eta,\sigma} \text{ and } h = \mathsf{e} \\
(\_,\eta,\sigma,h) \in [\![e_E \in e_S]\!]_\pi\, w &\text{ iff } [\![e_E]\!]_\eta \in [\![e_S]\!]_{\eta,\sigma} \text{ and } h = \mathsf{e} \\
(\_,\eta,\sigma,h) \in [\![e_E \notin e_S]\!]_\pi\, w &\text{ iff } [\![e_E]\!]_\eta \notin [\![e_S]\!]_{\eta,\sigma} \text{ and } h = \mathsf{e} \\
(\_,\eta,\sigma,h) \in [\![e^1_S \subseteq e^2_S]\!]_\pi\, w &\text{ iff } [\![e^1_S]\!]_{\eta,\sigma} \subseteq [\![e^2_S]\!]_{\eta,\sigma} \text{ and } h = \mathsf{e} \\
(\_,\eta,\_,h) \in [\![e_1 = e_2]\!]_\pi\, w &\text{ iff } [\![e_1]\!]_\eta = [\![e_2]\!]_\eta \text{ and } h = \mathsf{e} \\
(n,\eta,\sigma,h) \in [\![\mathrm{P}(\mathbf{e_V}\,\eta; \mathbf{e_S})]\!]_\pi\, w &\text{ iff } \mathrm{arity}_i(\mathrm{P}) = |\mathbf{e_V}| \text{ and } \mathrm{arity}_s(\mathrm{P}) = |\mathbf{e_S}| \\
&\qquad \text{and } (n,\eta,\sigma,h) \in \pi(\mathrm{P})([\![\mathbf{e_V}]\!]_\eta, [\![\mathbf{e_S}]\!]_{\eta,\sigma}) \\
[\![\mathsf{emp}]\!]_\pi\, w &= \mathrm{emp}\, w \\
[\![P \star Q]\!]_\pi\, w &= [\![P]\!]_\pi\, w \star [\![Q]\!]_\pi\, w \\
(\_,\eta,\_,h) \in [\![e_A \mapsto e^1_V, \ldots, e^n_V]\!]_\pi\, w &\text{ iff } \forall i \in \{0..n{-}1\}.\, [\![e_A]\!]_\eta + i \in \mathsf{dom}(h) \text{ and} \\
&\qquad \forall i \in \{0..n{-}1\}.\, h([\![e_A]\!]_\eta + i) = [\![e^i_V]\!]_\eta \\
(n,\eta,\sigma,\_) \in [\![\{P\}k(t_1,\ldots,t_n)\{Q\}]\!]_\pi\, w &\text{ iff } [\![k]\!]_\eta = \lceil \mathsf{proc}(z_1,\ldots,z_n)\{\mathsf{locals}\,\mathbf{y}; C\}\rceil \text{ and} \\
&\qquad w, \eta, \sigma \models^{\gamma'}_n ([\![P]\!]_\pi, \mathsf{call}\ \mathcal{N}(\mathbf{t}), [\![Q]\!]_\pi) \text{ where} \\
&\qquad \gamma' = \gamma[\mathcal{N} \mapsto (\mathsf{proc}(z_1,\ldots,z_n)\{\mathsf{locals}\,\mathbf{y}; C\})] \\
&\qquad \text{and } \mathcal{N} \notin \mathsf{dom}(\gamma) \\
[\![P \otimes Q]\!]_\pi\, w &= \big([\![P]\!]_\pi \otimes \iota([\![Q]\!]_\pi)\big)\, w
\end{aligned}
$$

Fig. 11 Interpretation of assertions.

Lemma 6 *Let I be a set and suppose that, for each $i \in I$, $F_i : \mathit{Pred}^I \to \mathit{Pred}$ is a contractive function. Then there exists a unique $\mathbf{p} = (p_i)_{i\in I} \in \mathit{Pred}^I$ such that $F_i(\mathbf{p}) = p_i$, for all $i \in I$.*

Since we use predicates with (integer and set) arguments, I needs to be chosen accordingly.

$$I = \sum_{\mathrm{P} \in \mathrm{PredName}} \mathbb{Z}^{\mathit{arity}_i(\mathrm{P})} \times \mathit{Set}^{\mathit{arity}_s(\mathrm{P})}$$

Since predicates cannot use global variables we thus get an interpretation for recursively defined predicates. The existence of recursively defined predicates is only guaranteed if their semantics is contractive which is in turn guaranteed if the right hand side of their declaration is an instance of the pattern discussed at length in [16].

Definition 2 (Soundness of Predicate Context) If Π is a list of (syntactic) predicate declarations then define its semantic validity w.r.t. a concrete predicate environment π as follows:

$$
\begin{aligned}
\pi \models \Pi \text{ iff } & (\mathsf{recdef}\ \mathrm{P}(\mathbf{x};\boldsymbol{\alpha}) := Q) \in \Pi \text{ implies } \pi(\mathrm{P}) \in \mathbb{Z}^{|\mathbf{x}|} \times \mathit{Set}^{|\boldsymbol{\alpha}|} \to \mathit{UPred}(H) \\
& \text{and for all } \mathbf{v} \in \mathbb{Z}^{|\mathbf{x}|}, \mathbf{s} \in \mathit{Set}^{|\boldsymbol{\alpha}|} \\
& \forall n, \eta, \sigma, h.\ (n,\eta,\sigma,h) \in \pi(\mathrm{P})(\mathbf{v},\mathbf{s}) \Leftrightarrow (n, \eta[\mathbf{x} \mapsto \mathbf{v}], \sigma[\boldsymbol{\alpha} \mapsto \mathbf{s}], h) \in [\![Q]\!]_\pi \\
& \text{and if additionally } (\mathsf{recdef}\ \mathrm{P}(\mathbf{x};\boldsymbol{\alpha}) := Q) \in \\
& \mathrm{LsegDefs}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \\
& \text{then for any R such that} \\
& \forall n, \eta, \sigma, h.\ (n,\eta,\sigma,h) \in \mathrm{R}(\mathbf{v},\mathbf{s}) \Leftrightarrow (n, \eta[\mathbf{x} \mapsto \mathbf{v}], \sigma[\boldsymbol{\alpha} \mapsto \mathbf{s}], h) \in [\![Q]\!]_\pi \\
& \text{we have that } \pi(\mathrm{P}) \subseteq \mathrm{R} \\
& \text{and} \\
& (\mathsf{recdef}\ \mathrm{P}(\mathbf{x};\boldsymbol{\alpha}) := \mathrm{R}(\mathbf{y};\boldsymbol{\beta}) \circ \Psi) \in \Pi \text{ implies} \\
& \pi(\mathrm{P}) \in \mathbb{Z}^{|\mathbf{x}|} \times \mathit{Set}^{|\boldsymbol{\alpha}|} \to \mathit{UPred}(H) \\
& \text{and for all } \mathbf{v} \in \mathbb{Z}^{|\mathbf{x}|}, \mathbf{s} \in \mathit{Set}^{|\boldsymbol{\alpha}|} \text{ and } n, \eta, \sigma, h \\
& (n,\eta,\sigma,h) \in \pi(\mathrm{P})(\mathbf{v},\mathbf{s}) \Leftrightarrow (n, \eta[\mathbf{x} \mapsto \mathbf{v}], \sigma[\boldsymbol{\alpha} \mapsto \mathbf{s}], h) \in \left[\!\!\left[\hat{S}_P^{\Psi} \right]\!\!\right]_\pi \\
& \text{where } \hat{S}_P^{\Psi} \text{ is as defined in Lemma 8 below.} \\
& \text{and} \\
& (\forall \mathrm{P}(\mathbf{x};\boldsymbol{\alpha})) \in \Pi \text{ implies } \pi(\mathrm{P}) \in \mathbb{Z}^{|\mathbf{x}|} \times \mathit{Set}^{|\boldsymbol{\alpha}|} \to \mathit{UPred}(H)
\end{aligned}
$$

This definition ensures that π satisfies the right conditions for predicate declarations, declarations with invariant extension, abstract declarations, and inductive declarations, respectively.

5.3 Semantics of judgements

First we define the semantics of assertions:

Definition 3 If Π is a predicate declaration and A an assertion then A is valid in Π iff $\forall \pi \models \Pi.\ [\![A]\!]_\pi = \top$. This validity judgement is abbreviated $\Pi \models A$. We sometimes drop Π to write simply $\models A$ if the predicate context is irrelevant.

Like the semantic judgement $\pi \models \Pi$ for predicate declaration, we also need one for procedure declarations.

Definition 4 If Γ is a list of (syntactic) procedure declarations with pre- and postconditions, then define the semantic validity for procedure environments as follows:

$$
\begin{aligned}
\gamma \models^n_\pi \Gamma \text{ iff } & (\mathsf{proc}\ \mathcal{F}(\mathbf{z})\ \forall \mathbf{x}, \boldsymbol{\alpha}.\ \mathsf{pre} : P\ \mathsf{post} : Q\ \{\mathsf{locals}\,\mathbf{y}; C\}) \in \Gamma \text{ implies} \\
& \gamma(\mathcal{F}) = \mathsf{proc}\ (\mathbf{z})\{\mathsf{locals}\,\mathbf{y}; C\} \text{ and } n \models^\gamma_\pi \{P\}\mathsf{call}\,\mathcal{F}(\mathbf{z})\{Q\} \qquad \text{and} \\
& (\mathsf{proc\ abstract}\ \mathcal{F}(\mathbf{z})\ \forall \mathbf{x}, \boldsymbol{\alpha}.\ \mathsf{pre} : P\ \mathsf{post} : Q) \in \Gamma \text{ implies} \\
& n \models^\gamma_\pi (P, \mathsf{call}\,\mathcal{F}(\mathbf{z}), Q).
\end{aligned}
$$

We write more succinctly $\gamma \models_\pi \Gamma$ if $\forall n \in \mathbb{N}.\, \gamma \models^n_\pi \Gamma$. The semantic version of judgement $\Pi; \Gamma \Vdash \{P\}\, C\, \{Q\}$ is defined as follows:

$$\Pi; \Gamma \models \{P\}\, C\, \{Q\} \text{ iff} \\ \forall n \in \mathbb{N}. \forall \pi \models \Pi. \forall \gamma \models^n_\pi \Gamma.\ n \models^\gamma_\pi (P, C, Q).$$

Assume now the module (procedure declarations) $(\mathcal{F}_i)_{1 \le i \le n}$ under consideration are summarised in a procedure declaration environment $\Gamma_\mathcal{F}$. Then the semantics of judgement $\Pi; \Gamma_\mathcal{A} \Vdash \{P\}\, \mathcal{F}(\mathbf{z})\, \{Q\}$ is defined as

$$\Pi; \Gamma_\mathcal{A} \models \{P\}\, \mathcal{F}(\mathbf{z})\, \{Q\} \text{ iff} \\ \forall \pi \models \Pi. \forall \gamma \models_\pi \Gamma_\mathcal{A}.\ \models^{\gamma \cup \rho}_\pi (P, \mathsf{call}\ \mathcal{F}(\mathbf{z}), Q).$$

where $\rho \models \Gamma_\mathcal{F}$.

5.4 Soundness of Assertion Logic

Our Hoare logic uses axioms for deriving assertions via entailment. Some of the more interesting ones have been collected in Figure A.2. We have to establish their soundness.

Theorem 1 (Soundness of $\Vdash$ for entailment of assertions) *The rules for $\Vdash$ given in Section 4.2 are all sound w.r.t. semantics given in Definition 4, in other words*

$$\Pi \Vdash P \Rightarrow Q \quad \textit{implies } \forall \pi \models \Pi. [\![P \Rightarrow Q]\!]_\pi = \top$$

Proof Since most of the usual logical reasoning is outsourced to the SMT solver which is assumed to be sound, we only have to deal with the more peculiar entailment axioms. The ones from Separation Logic are sound due to Lemma 1. The distribution laws for $\otimes$ are sound by the semantic definitions as given in Section 5.2.1. Rule (RUnique) holds in our model only assuming that the semantics of assertion R is contractive in predicate variable X as we then know that R gives rise to a fixpoint that is necessarily unique in ultrametric spaces. Rules (Split) and (Join) can be shown correct by induction (on the semantics) which itself holds since the semantics of predicates declared as *LSegDefs* admit induction. The (semantic) rule that we use for this is:

ListInduction

$$\frac{\begin{array}{c}\mathsf{recdef}\ \mathrm{L}(s,t;\alpha) := Q \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \\ \Pi, (\mathrm{L}(s,t;\alpha) \Leftrightarrow Q) \models s = t \star \alpha = \emptyset \Rightarrow P \\ \Pi, (\mathrm{L}(s,t;\alpha) \Leftrightarrow Q) \models \begin{pmatrix} P[s, \alpha \backslash n, \beta]\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \star \\ A_1 \star \cdots \star A_N\ \star\ \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \end{pmatrix} \Rightarrow P\end{array}}{\Pi, (\mathrm{L}(s,t;\alpha) \Leftrightarrow Q) \models \mathrm{L}(s,t;\alpha) \Rightarrow P}$$

where $n, \mathbf{v}, \beta \notin fv(P)$. Note that spatial implication is required to express the conclusions of those axioms as instances of the above induction scheme but we already know that this is available in *Pred*. For instance, for (Join) the predicate P needs to be instantiated as $\forall e_2, e_\beta. \mathrm{L}(t, e_2; e_\beta) \mathrel{-\!\!*} \mathrm{L}(s, e_2; \alpha \cup e_\beta)$. We could also *axiomatise spatial implication* in the assertion logic which would allows us to *derive* (Split) and (Join) from (ListInduction) but since our verifier does not support

spatial implication there is actually no need for this.
The special interpretation of pure assertions guarantees axiom ($\star$-SplitPureLeft). To see this assume $(n,\eta,\sigma,h) \in [\![A \star \Phi]\!]_\pi$. Then h has the form $h_1 \cdot h_2$ where $(n,\eta,\sigma,h_1) \in [\![A]\!]_\pi$ and $(n,\eta,\sigma,h_2) \in [\![\Phi]\!]_\pi$. As Φ is pure $h_2 = \mathsf{e}$ and so $h = h_1$ such that we get the desired result.
The axiom ($\star$-SplitPureRight) also holds due to the special interpretation of pure assertions. To see this assume $(n,\eta,\sigma,h) \in [\![A \wedge (\Phi \star \mathit{true})]\!]_\pi$. Then $(n,\eta,\sigma,h) \in [\![A]\!]_\pi$ and $(n,\eta,\sigma,h) \in [\![\Phi \star \mathit{true}]\!]_\pi$. Thus h has the form $h_1 \cdot h_2$ where $(n,\eta,\sigma,h_1) \in [\![\Phi]\!]_\pi$ and $(n,\eta,\sigma,h_2) \in [\![\mathit{true}]\!]_\pi$. Since Φ is pure we have $h_1 = \mathsf{e}$ and so $h = h \cdot h_1$, and from this $(n,\eta,\sigma,h) \in [\![A \star \Phi]\!]_\pi$ follows.
(ShallowFrameProcedures) and (ShallowFrame) hold as framing is "baked into" the definition of the semantics of triples.

5.5 Soundness of Hoare Rules

Theorem 2 (Soundness of $\Vdash$ for triples) *The rules for $\Vdash$ given in Section 4.1 are all sound w.r.t. semantics given in Definition 4.*

Proof Largely by adapting the proofs in [40, 7], adding the procedure and predicate environments appropriately. We discuss the new or substantially changed rules:
Rule (RecursiveProcedures) : For a given $\pi \models \Pi$ and $\gamma \models_\pi \Gamma_{\mathcal{A}}$ and a procedure declaration environment $\rho \models_\pi \Gamma_{\mathcal{F}}$ that contains all n non-abstract declared procedures $\mathcal{F}_i$ we show for all i with $1 \le i \le n$ simultaneously that

$$\models_\pi^{\gamma \cup \rho} (\mathit{pre}(\mathcal{F}_i), \mathsf{call}\ \mathcal{F}_i(\mathit{params}(\mathcal{F}_i)), \mathit{post}(\mathcal{F}_i))$$

which is equivalent to showing that

$$k \models_\pi^{\gamma \cup \rho} (\mathit{pre}(\mathcal{F}_i), \mathsf{call}\ \mathcal{F}_i(\mathit{params}(\mathcal{F}_i)), \mathit{post}(\mathcal{F}_i))$$

for all k or equivalently for all η, σ, w, k:

$$w,\eta,\sigma \models_k^{\gamma \cup \rho} ([\![\mathit{pre}(\mathcal{F}_i)]\!]_\pi, \mathsf{call}\ \mathcal{F}_i(\mathit{params}(\mathcal{F}_i)), [\![\mathit{post}(\mathcal{F}_i)]\!]_\pi) \tag{5}$$

This is shown by induction on k. By definition of the operational semantics for $k = 0$ the claim holds automatically as (more than) one step is used for interpreting the call statement so it cannot reduce to skip in zero steps. Assume we have proved the claim (5) for all number less than k and show it for k: According to Definition 1 of triple semantics, assume $m < k$, $h \in \mathit{Heap}$, and $r \in \mathit{UPred}$ and let $(m,\eta,\sigma,h) \in [\![\mathit{pre}(\mathcal{F}_i)]\!]_\pi\, w \star i^{-1}(w)(\mathrm{emp}) \star r$. Then it must be that $h = h_1 \cdot h_2 \cdot h_3$, where

(a) $(m,\eta,\sigma,h_1) \in [\![\mathit{pre}(\mathcal{F}_i)]\!]_\pi\, w$
(b) $(m,\eta,\sigma,h_2) \in \iota^{-1}(w)(\mathrm{emp})$ and
(c) $(m,\eta,\sigma,h_3) \in r$.

First we need to show that $(\mathsf{call}\ \mathcal{F}_i(\mathit{params}(\mathcal{F}_i)), s \cdot \eta, h) \in \mathrm{Safe}_m^{\gamma \cup \rho}$ for any stack s. By definition of the operational semantics – as the unfolding of a procedure and executing the additional _ ; return statement take three steps – it suffices to show that

$$(\mathit{body}(\mathcal{F}_i), s \cdot \eta \cdot \eta[\mathit{locals}(\mathcal{F}_i) \mapsto \mathbf{0}], h) \in \mathrm{Safe}_{m-3}^{\gamma \cup \rho}$$

which follows from the rule's hypothesis as follows: First we know by assumption that $\gamma \models_\pi^{m-3} \Gamma_\mathcal{A}$; by induction hypothesis $\rho \models_\pi^{m-3} \Gamma_\mathcal{F}$ so $\gamma \cup \rho \models^{m-3} \Gamma_\mathcal{A}, \Gamma_\mathcal{F}$, thus by the rule's hypothesis $m-3 \models_\pi^{\gamma\cup\rho} (pre(\mathcal{F}_i), body(\mathcal{F}_i), post(\mathcal{F}_i))$. Now from this, (a-c), and the fact that $locals(\mathcal{F}_i)$ do not appear in $pre(\mathcal{F}_i)$ the required follows. Secondly, we need to show for all $k \leq m, h' \in Heap, \eta' \in Env$, if

$$(\mathsf{call}\ \mathcal{F}_i(params(\mathcal{F}_i)), s \cdot \eta, h) \rightsquigarrow_k^\gamma (\mathsf{skip},\ s \cdot \eta', h') \tag{6}$$

where $\eta' = \eta$ since procedures in our language only have side effects on the heap, then $(m-k, \eta', \sigma, h') \in [\![post(\mathcal{F}_i)]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$. By (6) and definition of operational semantics we know that

$$(body(\mathcal{F}_i), s \cdot \eta \cdot \eta[locals(\mathcal{F}_i) \mapsto \mathbf{0}], h) \rightsquigarrow_{k-3}^\gamma (\mathsf{skip},\ s \cdot \eta \cdot \eta'', h'') \tag{7}$$

must hold for some $h'' \in Heap$ and $\eta'' \in Env$. By the same argument as for the first condition we obtain

$$k-3 \models_\pi^{\gamma\cup\rho} (pre(\mathcal{F}_i), body(\mathcal{F}_i), post(\mathcal{F}_i)) \tag{8}$$

As before, from (8), using (a-c) and the fact that $pre(\mathcal{F}_i)$ does not contain any of the local variables of $\mathcal{F}_i$ we obtain that $(m-(k-3), \eta'', \sigma, h'') \in [\![post(\mathcal{F}_i)]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$ and by downward closure that $(m-k, \eta'', \sigma, h'') \in [\![post(\mathcal{F}_i)]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$. But by the definition of the operational semantics we know that $h'' = h'$, we already know $\eta' = \eta$, and that $\eta'' = \eta[locals(\mathcal{F}_i) \mapsto \ldots]$ since we do not allow assignments to formal parameters. Since $post(\mathcal{F}_i)$ does not contain any of the local variables of $\mathcal{F}_i$ from this follows that $(m-k, \eta', \sigma, h') \in [\![post(\mathcal{F}_i)]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$.

Rule (Call) : Assume that $\pi \models \Pi$ and for all n, η, σ, h that

$$\forall m \leq n.\, (m, \eta, \sigma, h) \in [\![\forall \mathbf{x}.\, \{A\}\, k(\mathbf{p})\, \{B\})]\!]_\pi \Rightarrow (m, \eta, \sigma, h) \in [\![\{P\}\, k(\mathbf{e})\, \{Q\}]\!]_\pi \tag{9}$$

Further assume an n and γ such that $\gamma \models_\pi^n \Sigma, \{A\}\, \mathcal{F}(\mathbf{p})\, \{B\}$. From this we know by definition that

$$n \models_\pi^\gamma (A, \mathsf{call}\, \mathcal{F}(\mathbf{p}), B) \tag{10}$$

We need to show that $n \models_\pi^\gamma (P, \mathsf{call}\, \mathcal{F}(\mathbf{e}), Q)$. Assuming arbitrary w, η, σ it suffices to prove that $w, \eta, \sigma \models_n^\gamma ([\![P]\!]_\pi, \mathsf{call}\, \mathcal{F}(\mathbf{e}), [\![Q]\!]_\pi)$. From (9) with $m := n$, $\eta' = \eta[k \mapsto \lceil \gamma(\mathcal{F}) \rceil]$ and $\gamma' := \gamma[\mathcal{N} \mapsto \gamma(\mathcal{F})]$ we obtain

$$(\forall \mathbf{v}.\, w, \eta'[\mathbf{x} \mapsto \mathbf{v}], \sigma \models_n^{\gamma'} ([\![A]\!]_\pi, \mathsf{call}\, \mathcal{N}(\mathbf{p}), [\![B]\!]_\pi)) \Rightarrow w, \eta', \sigma \models_n^{\gamma'} ([\![P]\!]_\pi, \mathsf{call}\, \mathcal{N}(\mathbf{e}), [\![Q]\!]_\pi) \tag{11}$$

By definition of γ' and η', freshness of k and $\mathcal{N}$, setting $\mathbf{v} := \eta(\mathbf{x})$ (11) entails

$$w, \eta, \sigma \models_n^\gamma ([\![A]\!]_\pi, \mathsf{call}\, \mathcal{F}(\mathbf{p}), [\![B]\!]_\pi) \Rightarrow w, \eta, \sigma \models_n^\gamma ([\![P]\!]_\pi, \mathsf{call}\, \mathcal{F}(\mathbf{e}), [\![Q]\!]_\pi)$$

which by (10) and definition of $n \models_\pi^\gamma (P, \mathsf{call}\, \mathcal{F}(\mathbf{e}), Q)$ completes the proof.

Rule (StoreProc) : Assume $\pi \models \Pi$ and $\gamma \models_\pi^n \{A\}\, \mathcal{F}(\mathbf{p})\, \{B\}$. From this we get:

$$n \models_\pi^\gamma (A, \mathsf{call}\ \mathcal{F}(\mathbf{p}), B) \tag{12}$$

We need to show that $n \models_\pi^\gamma (e \mapsto \_, [e] := \mathcal{F}(\mathbf{t}), e \mapsto S(\cdot))$ where

$$S(\cdot) = (\forall \mathbf{p}|_U, \mathbf{x}.\ \{A\} \cdot (\mathbf{p}|_U)\, \{B\})\, [\mathbf{p}|_{I\setminus U} \backslash \mathbf{t}|_{I\setminus U}]$$

where $|\mathbf{t}| = |\mathbf{p}|$, t_i either value expression a_i or $\_$; $\mathbf{x} = fv(A,B) - \mathbf{p}$ and $\mathbf{p} = (p_i)_{i\in I}$; $U = \{i \in I \mid t_i = \_\}$ $\mathbf{p}|_X = (p_i)_{i\in I\cap X}$. According to Definition 1 of $\models$, assume $m < n$ and $r \in \mathit{UPred}$ and let $(m,\eta,\sigma,h) \in [\![e \mapsto \_]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$. Then it must be that $h = h_1 \cdot h_2 \cdot h_3$, where

(a) $(m,\eta,\sigma,h_1) \in [\![e \mapsto \_]\!]_\pi\, w$
(b) $(m,\eta,\sigma,h_2) \in \iota^{-1}(w)(\mathrm{emp})$ and
(c) $(m,\eta,\sigma,h_3) \in r$.

It is easy to show now that $([e] := \mathcal{F}(\mathbf{t}), s\cdot\eta, h) \in \mathrm{Safe}_1^\gamma$ (so $m = 1$) because $[\![e]\!]_\eta \in \mathsf{dom}(h)$ due to (a), $\mathcal{F} \in \mathsf{dom}(\gamma)$ by assumption and $\mathit{papply}\ \gamma(\mathcal{F})(\mathbf{u})$ is defined where $u_i = \_$ if $t_i = \_$ and $u_i = \eta(x_i)$ if $t_i = x_i$. It remains to show that $(m-1,\eta,\sigma,h_1[[\![e]\!]_\eta \mapsto \lceil \mathit{papply}\ \gamma(\mathcal{F})\ (u_1,\ldots,u_n)\rceil]) \in [\![e \mapsto S(\cdot)]\!]_\pi\, w$. By definition of assertion semantics it suffices to show for all $\mathbf{v}$,$\mathbf{v'}$ that

$$w,\eta[\mathbf{x}\mapsto\mathbf{v},\mathbf{p}|_\mathbf{U}\mapsto\mathbf{v'}],\sigma \models^{\gamma'}_{m-1} (A[\mathbf{p}|_{I\setminus U}\backslash\mathbf{t}|_{I\setminus U}], \mathsf{call}\,\mathcal{N}(\mathbf{p}|_U), B[\mathbf{p}|_{I\setminus U}\backslash\mathbf{t}|_{I\setminus U}])$$

where $\gamma' = \gamma[\mathcal{N} \mapsto (\mathit{papply}\ \gamma(\mathcal{F})\ (u_1,\ldots,u_n))]$. Since $\eta(\mathbf{t}|_{I\setminus U}) = \mathbf{u}$ (the arguments of *papply*) and by definition of γ' this follows from

$$w,\eta[\mathbf{x}\mapsto\mathbf{v},\mathbf{p}|_\mathbf{U}\mapsto\mathbf{v'},\mathbf{p}|_{I\setminus U}\mapsto\mathbf{t}|_{I\setminus U}],\sigma \models^{\gamma'}_{m-1} (A, \mathsf{call}\,\mathcal{F}(\mathbf{p}), B)$$

which is obtained from (12) by instantiation and downward closure (as $m-1 < m < n$).

Rule (Eval) : We must show that assuming

$$\Pi;\Gamma \models P \Rightarrow e \mapsto \{P\}\cdot(\mathbf{e})\{Q\} \star \mathit{true} \tag{13}$$

it holds that $\Pi;\Gamma \models \{P\}\mathsf{eval}\ [e](\mathbf{e})\{Q\}$. So let $\pi \models \Pi$, $\gamma \models_\pi \Gamma$, and let further η be an integer variable environment, σ be a set variable environment and $w \in W$ and $n \in \mathbb{N}$. We must show that

$$w,\eta,\sigma \models^\gamma_n ([\![P]\!]_\pi, \mathsf{eval}\ [e](\mathbf{e}), [\![Q]\!]_\pi) \tag{14}$$

According to Definition 1, assume $m < n$, $h \in \mathit{Heap}$, $r \in \mathit{UPred}$ and let $(m,\eta,\sigma,h) \in [\![P]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$. Then it must be that $h = h_1 \cdot h_2 \cdot h_3$, where

(a) $(m,\eta,\sigma,h_1) \in [\![P]\!]_\pi\, w$
(b) $(m,\eta,\sigma,h_2) \in \iota^{-1}(w)(\mathrm{emp})$ and
(c) $(m,\eta,\sigma,h_3) \in r$.

From (a), (13) and the semantics of implication we obtain

$$(m,\eta,\sigma,h_1) \in [\![e \mapsto \{P\}\cdot(\mathbf{e})\{Q\} \star \mathit{true}]\!]_\pi\, w \tag{15}$$

and thus there are heaps $[l \mapsto v]$ with $[\![e]\!]_\eta = l$ and h_r such that $h_1 = [l \mapsto v]\cdot h_r$ and $(m,\eta,\sigma,[l\mapsto v]) \in [\![e \mapsto \{P\}\cdot(\mathbf{e})\{Q\}]\!]_\pi\, w$ from which follows that

$$v \equiv \lceil \mathsf{proc}(\mathbf{z})\{\mathsf{locals}\, y; C\}\rceil \text{ and } w,\eta,\sigma \models^{\gamma'}_m ([\![P]\!]_\pi, \mathsf{call}\ \mathcal{N}(\mathbf{e}), [\![Q]\!]_\pi) \tag{16}$$

where $\gamma' = \gamma[\mathcal{N} \mapsto \mathsf{proc}(\mathbf{z})\{\mathsf{locals}\, y; C\}]$ for a fresh procedure name $\mathcal{N}$.
We observe by definition of the semantics, the fact that $h = [l\mapsto v]\cdot h_r \cdot h_2 \cdot h_3$ and $\gamma'(\mathcal{N}) = \lceil v\rceil^{-1}$ (due to the first part of (16)), that for all stacks s and all $k \in \mathbb{N}$

$$(\mathsf{eval}\ [e](\mathbf{e}), s\cdot\eta, h) \rightsquigarrow^\gamma_k \Delta \iff (\mathsf{call}\ \mathcal{N}(\mathbf{e}), s\cdot\eta, h) \rightsquigarrow^{\gamma'}_k \Delta \tag{17}$$

First, we have to show for any stack s that $(\mathsf{eval}\ [e](\mathbf{e}), s \cdot \eta, h) \in \mathrm{Safe}_m^\gamma$. But (17) tells us that this is the case iff $(\mathsf{call}\ \mathcal{N}(\mathbf{e}), s \cdot \eta, h) \in \mathrm{Safe}_m^{\gamma'}$ which, in turn, follows from the second part of (16) and the fact that $(m, \eta, \sigma, h) \in [\![P]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$ by (a-c).
Next we show the second condition. Thus, assume $k \leq m$ and

$$(\mathsf{eval}\ [e](\mathbf{e}), s \cdot \eta, h) \rightsquigarrow_k^\gamma (\mathsf{skip},\ s \cdot \eta', h') \tag{18}$$

We need to show that $(m-k, \eta', \sigma, h') \in [\![Q]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$. Again, by (17) we know that $(\mathsf{call}\ \mathcal{N}(\mathbf{e}), s \cdot \eta, h) \rightsquigarrow_k^\gamma (\mathsf{skip},\ s \cdot \eta', h')$ from which by (16) it follows that $(m-k, \eta', \sigma, h') \in [\![Q]\!]_\pi\, w \star \iota^{-1}(w)(\mathrm{emp}) \star r$ which completes the proof.

Other interesting rules: rule (RUnique) is not sound in general. However it does hold if the predicates in π are unique solutions of their defining equations in Π. This can be guaranteed together with the existence of an $\pi \in \Pi$ if the recursive predicate declarations are of a pattern described in detail in [16]. The ghost statements are all interpreted like skip; they do not have a computational effect and the soundness of the corresponding rules can thus be shown relatively straightforwardly using the consequence rule.

5.5.1 Soundness of the SMT solver

The following theorem states that the judgement proven by the SMT solver is sound for our logic (assuming that the solver is already sound for classical logic). We already have a function that can eliminate spatial parts from assertions, we also assume we have an operation $\widehat{\_}$ that replaces $\star$ by $\wedge$ such that $\widehat{purify(P)}$ is a classical statement about numbers, tuples and sets.

Theorem 3 (Soundness of SMT) *Assume the SMT solver correctly solves entailments of translated pure assertions and that Φ and Θ are pure assertions. Then $\Phi \vdash_{SMT} \Theta$ implies $[\![\Phi \Rightarrow \Theta]\!]_\pi = \top$ for all predicate environments π.*

Proof By the assumption and the correctness of the SMT solver we obtain $\vdash_{cl} \widehat{\Phi} \Rightarrow \widehat{\Theta}$ (†). We prove two lemmas:

1. For all pure assertions Υ, and all $w \in W, n \in \mathbb{N}, h \in \mathit{Heap}, \eta \in \mathit{Env}, \sigma \in \mathit{SetEnv}$ it holds that $(n, \eta, \sigma, h) \in [\![\Upsilon]\!]_\pi\, w$ implies $h = \mathsf{e}$ and $\vdash_{cl} \widehat{\Upsilon}$
2. For all pure assertions Υ, if $\vdash_{cl} \widehat{\Upsilon}$ then $(n, \eta, \sigma, \mathsf{e}) \in [\![\Upsilon]\!]_\pi\, w$ for all $w \in W, n \in \mathbb{N}, \eta \in \mathit{Env}, \sigma \in \mathit{SetEnv}$.

With the two lemmas we can now show the theorem. Assume we have $(m, \eta, \sigma, h) \in [\![\Phi]\!]_\pi\, w$. Then by (1) we get that $h = \mathsf{e}$ and $\vdash_{cl} \widehat{\Phi}$, from which by (†) it follows that $\vdash_{cl} \widehat{\Theta}$ and finally by (2) that $(m, \eta, \sigma, \mathsf{e}) \in [\![\Theta]\!]_\pi\, w$ concluding the proof.
Lemmas (1) and (2) can be shown by induction on the structure of assertion (disjunct) Φ and holds by definition of the semantics where it can be seen that assertions that should be independent of the heap only hold in the empty heap e.

Note that a pure formula Υ in our logic does not hold in the common sense (where its interpretation would need to hold for all heaps), but just for the empty heap. This is why we do not have *true* as a pure assertion and thus do not have *true* in the low level assertion logic. This is not a problem as we are interested in proving validity of Hoare triples and not assertions and since pure assertions will always be "framed onto" spatial assertions.

6 Automation of program verification

In this section we explain how the automated verification works by giving deterministic rules for proof search and discuss their soundness.

6.1 Overview

The automatic prover consists of three main parts.

1. Verification condition generator: The verification condition (VC) generator reads in annotated programs and produces from them a set of VCs, such that if all the VCs hold then the input program meets its specifications. Each VC is of the form $\Pi; \Gamma \Vdash \{P\}\, C\, \{Q\}$ as explained earlier.

2. Symbolic execution engine: We prove Hoare triples using symbolic execution with separation logic, based on ideas put forward in [5] and now well established. The symbolic execution algorithm relies on automatic entailment provers at various points. At the end of each symbolic execution step happens a cleanup operation that is of a "cosmetic nature", ie. its purpose is to keep the goals as small as possible without changing their semantics.

3. Entailment provers: The use of nested triples adds considerably to the difficulty of proving entailments automatically. Because assertions can contain triples and vice versa, we need solvers for entailments between assertions and triples, respectively, defined mutually recursively.

In fact in our implementation there are proof systems for five different judgements; many of these proof systems need to invoke each other. The judgements and their informal meanings are as follows. Shaded variables (such as the frame Θ) are those whose value is not given as an input to the prover, but rather is inferred by the proof rules.

- $\Phi \vdash^{I} \exists \mathbf{v}.\Upsilon \star \Theta$. Entailment between assertion disjuncts. Spatial conjunction Φ entails assertion disjunct $\exists \mathbf{v}.\Upsilon$ with frame Θ left over. I is a mapping from the existentially quantified variables $\mathbf{v}$ to appropriate witnesses for these.
- $B_1 \vdash B_2$. Entailment between behavioural specifications.
- $B \vdash_{\text{find-post}} \{\Phi\} \cdot (\mathbf{t}) \left\{ Q \right\}$. Computing the postcondition for an invocation. This judgement computes an assertion Q which describes the state that results from invoking code with behaviour B in a state described by spatial conjunction Φ.
- $\Pi : \Upsilon \vdash_{\text{find-tr}} \exists\, \mathbf{v}\, .\, e_A \mapsto B \star R^{pure}$.[10] Finding specifications for code stored on the heap inside a symbolic state. R^{pure} includes all the pure formulae that are the result of any unfolding or splitting required during the search and that are relevant for $B(\cdot)$ and $\mathbf{x}$ are the existentially quantified variables obtained from unfolding or splitting.
- $\Pi, \Gamma \triangleright \{P\}\, C\, \{Q\}$. Symbolic execution.

We shall show that our proof systems for these judgements are all sound. The following theorem shows soundness of the (implemented) symbolic execution, en-

[10] In [14] we had a less refined version ignoring pure facts R^{pure} about variables $\mathbf{x}$ obtained from unfolding which are important for certain examples, see Figure 12.

tailment and other proof search rules with respect to the "core" logic introduced in Section 5.5 which we already know are sound (see Theorem 2).

Theorem 4 Grand soundness theorem. *Our five proof systems are sound, that is:*

1. *If* $\Phi \vdash^{I} \exists \mathbf{v}.\Upsilon \star \Theta$ *(where* $fv(\Phi) \cap \mathbf{v} = \emptyset$*) then* $\models \Phi \Rightarrow \Upsilon[\mathbf{v}\backslash I(\mathbf{v})] \star \Theta$ *where:* $fv(\Theta) \subseteq fv(\Phi)$*,* $dom(I) = \mathbf{v}$ *and* $fv(Im(I)) \subseteq fv(\Phi)$*.*
2. *If* $B_1 \vdash B_2$ *then* $\models B_1[\_\backslash k] \Rightarrow B_2[\_\backslash k]$ *where* $k \notin fv(B_1, B_2)$*.*
3. *If* $B \vdash_{\mathit{find\text{-}post}} \{\Phi\} \cdot (\mathbf{t}) \{Q\}$ *then* $\models B[\_\backslash k] \Rightarrow \{\Phi\}\, k(\mathbf{t})\, \{Q\}$ *where* $k \notin fv(B, \Phi, Q)$*.*
4. *If* $\Pi : \Upsilon \vdash_{\mathit{find\text{-}tr}} \exists \mathbf{v}.\, e_A \mapsto B \star R^{pure}$ *then* $\Pi \models \Upsilon \Rightarrow \exists \mathbf{v}.\, e_A \mapsto B \star R^{pure} \star \Upsilon'$ *for some* Υ'*.*[11]
5. *If* $\Pi, \Gamma \triangleright \{P\}\, C\, \{Q\}$ *then* $\Pi; \Gamma \models \{P\}\, C\, \{Q\}$ *(that is, our symbolic execution rules are sound).*

We will also need the soundness of the SMT solver as shown in Theorem 3.

*Proof (**Grand Soundness Theorem**)* We have already shown in Section 5.5 the soundness of the Hoare logic rules for the rule for judgements $\Pi; \Gamma \models \{P\}\, C\, \{Q\}$ and $\Pi \Vdash A$. So in the soundness proofs we can freely use these "core" rules as listed in Appendix A. In other appendices we present detailed soundness arguments for selected interesting rules of each of the five judgements:

1. $\Phi \vdash^{I} \Theta$: See Appendix D where soundness of rules (InstUsingEq), (CancelPt1), and (InstMatchAddr) is shown.
2. $B_1 \vdash B_2$: See Appendix E where soundness of rules (RemoveRight), (DisjPre), (ExistsPre), and (TripleEnt) is shown.
3. $B \vdash_{\text{find-post}} \{\Phi\} \cdot (\mathbf{t}) \{Q\}$: See Appendix F where soundness of rules (InstParam), and (InferSpecForCall) is shown.
4. $\Pi \models \Upsilon \Rightarrow \exists \mathbf{v}.\, e_A \mapsto B \star R^{pure} \star \Upsilon'$: See Appendix H where soundness of rules (Find), (FindUnfold) and (FindSplit) is shown.
5. $\Pi, \Gamma \triangleright \{P\}\, C\, \{Q\}$: See Appendix C where soundness of rules (Lookup), (Eval) and (StoreCode) is shown.

6.2 Verification Condition generation

First the predicate context Π is built. This is done by collecting all predicate declarations provided by the user, and then adding some extra equivalences to allow the convenient folding and unfolding of predicates defined by invariant extension. More precisely, for each definition $\mathsf{recdef}\ \mathrm{S}(\mathbf{a}, \mathbf{b}) := \mathrm{P}(\mathbf{a}) \circ \Psi$ where P is recursively defined as $\mathsf{recdef}\ \mathrm{P}(\mathbf{a}) := R[\mathrm{P}]$ for some appropriate assertion R we automatically declare a predicate $\hat{\mathrm{S}}^{\Psi}_{\mathrm{P}}(\mathbf{a}, \mathbf{b})$ such that $\hat{\mathrm{S}}^{\Psi}_{\mathrm{P}}(\mathbf{a}, \mathbf{b}) \Leftrightarrow \mathrm{S}(\mathbf{a}, \mathbf{b})$. The details can be found in the following Section 6.2.1 (see Lemma 8). Note also that we assume that all existentially quantified variables in predicate definitions are renamed to be fresh in a way such that they can never clash with any other variables names in the future. This removes some side conditions from rules involving predicates and also allows the computation of the automatically declared predicates discussed above.

[11] We deviate from [14] here in the sense that we only require $\Rightarrow$ and not $\Leftrightarrow$. This is necessary to obtain a sound splitting rule for list segments not discussed in [14]. As a further advantage it allows one to use a "core" (Eval) rule more akin to the one used in proof search.

Secondly the procedure context Γ is built by collecting the specifications (pre- and postconditions) declared for each procedure (including abstract ones); for each procedure $\mathcal{F}$ we include in Γ the triple $\{pre(\mathcal{F})\}\,\mathcal{F}(params(\mathcal{F}))\,\{post(\mathcal{F})\}$.

Finally for each (non-abstract) procedure $\mathcal{F}$ we generate the following VC:

$$\Pi, \Gamma \triangleright \{pre(\mathcal{F})\}\ body(\mathcal{F})\ \{post(\mathcal{F})\}$$

which will be proved with the help of symbolic execution rules as described in Subsection 6.3

6.2.1 Invariant extension involving recursive predicates

Lemma 7 *For every recursively defined predicate* $\mathsf{recdef}\ \mathrm{P}(\mathbf{x};\boldsymbol{\alpha}) := R[\mathrm{P}]$ *and* Ψ *with free integer variables in* $\mathbf{y} \cup \mathbf{x}$*, free set variables in* $\boldsymbol{\beta} \cup \boldsymbol{\alpha}$ *none of which are existentially quantified in* R*, there is a recursive predicate definition* $\mathsf{recdef}\ \mathrm{S}^{\Psi}_{\mathrm{P}}(\mathbf{x},\mathbf{y};\boldsymbol{\alpha},\boldsymbol{\beta}) := Q(R)[\mathrm{S}^{\Psi}_{\mathrm{P}}]$ *such that* $\mathrm{S}^{\Psi}_{\mathrm{P}}(\mathbf{x},\mathbf{y};\boldsymbol{\alpha},\boldsymbol{\beta}) \Leftrightarrow \mathrm{P}(\mathbf{x};\boldsymbol{\alpha}) \otimes \Psi$.

Proof Plotkin's Lemma states that $h(\operatorname{fix} f) = \operatorname{fix} g$ if $g \circ h = h \circ f$ for $h : A \to B$ and endomaps $f : A \to A$ and $g : B \to B$. This can be easily shown by using the fixpoint property, $\operatorname{fix} f = f(\operatorname{fix} f)$, using the rule (RUnique). We will use this lemma, setting $h(P) := P \otimes \Psi$, $f(P) := R[P]$ and $g(P) := Q(R)[P]$ where we extend $\otimes$ to work on predicates with arguments of type $Pred^I$ in the natural pointwise fashion. It then only remains to show that

$$Q(R)[\_ \otimes \Psi] \Leftrightarrow R[\_] \otimes \Psi \tag{19}$$

to conclude that fix $Q(R) \Leftrightarrow (\text{fix } R) \otimes \Psi$. Define $Q(R)$ by induction on R accordingly:

$$\begin{aligned}
Q(R) &= _(\mathbf{e};\mathbf{s}) \text{ if } R = _(\mathbf{e};\mathbf{s}) \\
Q(R) &= \phi \text{ if } R = \phi \text{ and } \phi \text{ is an atomic pure formula} \\
Q(R) &= Q(R_1) \star Q(R_2) \text{ if } R = R_1 \star R_2 \\
Q(R) &= Q(R_1) \vee Q(R_2) \text{ if } R = R_1 \vee R_2 \\
Q(R) &= \forall x.Q(R_1) \text{ if } R = \forall x.R_1 \\
Q(R) &= \exists x.Q(R_1) \text{ if } R = \exists x.R_1 \\
Q(R) &= Q(R_1) \Rightarrow Q(R_2) \text{ if } R = R_1 \Rightarrow R_2 \\
Q(R) &= \{Q(R_1) \star \Psi\}\, e(\mathbf{t})\, \{Q(R_2) \star \Psi\} \text{ if } R = \{R_1\}\, e(\mathbf{t})\, \{R_2\}
\end{aligned}$$

We need to show (19) which is relatively straightforward. We provide details for some interesting cases.
$R = \_(\mathbf{e};\mathbf{s})$:

$$\begin{aligned}
Q(R)[_ \otimes \Psi] &= (_ \otimes \Psi)(\mathbf{e};\mathbf{s}) \\
&\Leftrightarrow_{\text{pointwise def. of } \otimes} (_(\mathbf{e};\mathbf{s}) \otimes \Psi) \\
&= R[_] \otimes \Psi
\end{aligned}$$

$R = \phi : Q(R)[\_ \otimes \Psi] = \phi \Leftrightarrow \phi \otimes \Psi = R[\_] \otimes \Psi$
$R = R_1 \star R_2$:

$$\begin{aligned} Q(R)[\_ \otimes \Psi] &= (Q(R_1) \star Q(R_2))[\_ \otimes \Psi] \\ &= Q(R_1)[\_ \otimes \Psi] \ \star \ Q(R_2)[\_ \otimes \Psi] \\ &\Leftrightarrow_{\text{Ind.Hypothesis}} (R_1[\_] \otimes \Psi) \ \star \ (R_2[\_] \otimes \Psi) \\ &\Leftrightarrow (R_1 \star R_2)[\_] \otimes \Psi \end{aligned}$$

$R = \{R_1\}\, e(\mathbf{t})\, \{R_2\}$:

$$\begin{aligned} Q(R)[\_ \otimes \Psi] &= \{Q(R_1) \star \Psi\}\, e(\mathbf{t})\, \{Q(R_2) \star \Psi\}\, [\_ \otimes \Psi] \\ &= \{Q(R_1)[\_ \otimes \Psi] \star \Psi\}\, e(\mathbf{t})\, \{Q(R_2)[\_ \otimes \Psi] \star \Psi\} \\ &\Leftrightarrow_{\text{Ind.Hypothesis}} \{R_1[\_] \otimes \Psi \star \Psi\}\, e(\mathbf{t})\, \{R_2[\_] \otimes \Psi \star \Psi\} \\ &\Leftrightarrow \{R_1[\_] \circ \Psi\}\, e(\mathbf{t})\, \{R_2[\_] \circ \Psi\} \\ &\Leftrightarrow \{R_1[\_]\}\, e(\mathbf{t})\, \{R_2[\_]\} \otimes \Psi \end{aligned}$$

$R = \exists x.\, R_1$:

$$\begin{aligned} Q(R)[\_ \otimes \Psi] &= \exists x. Q(R_1)[\_ \otimes \Psi] \\ &\Leftrightarrow_{\text{Ind.Hypothesis}} \exists x. (R_1[\_] \otimes \Psi) \\ &\Leftrightarrow (\exists x.\, R_1[\_]) \otimes \Psi \qquad \text{since } x \notin fv(\Psi) \end{aligned}$$

Lemma 8 *For every recursively defined predicate* recdef $\mathrm{P}(\mathbf{x}; \boldsymbol{\alpha}) := \mathrm{R}[\mathrm{P}]$ *and* Ψ *with free integer variables in* $\mathbf{y} \cup \mathbf{x}$ *and free set variables in* $\boldsymbol{\beta} \cup \boldsymbol{\alpha}$ *none of which are existentially quantified in R, there is a recursive predicate definition* recdef $\hat{\mathrm{S}}^{\Psi}_{\mathrm{P}}(\mathbf{x}, \mathbf{y}; \boldsymbol{\alpha}, \boldsymbol{\beta}) := \hat{\mathrm{R}}[\mathrm{S}^{\Psi}_{\mathrm{P}}]$ *such that* $\hat{\mathrm{S}}^{\Psi}_{\mathrm{P}}(\mathbf{x}, \mathbf{y}; \boldsymbol{\alpha}, \boldsymbol{\beta}) \Leftrightarrow \mathrm{P}(\mathbf{x}; \boldsymbol{\alpha}) \circ \Psi$.

Proof By Lemma 7 we can define $\hat{\mathrm{S}}^{\Psi}_{\mathrm{P}}(\mathbf{x}, \mathbf{y}; \boldsymbol{\alpha}, \boldsymbol{\beta}) := \mathrm{S}^{\Psi}_{\mathrm{P}}(\mathbf{x}, \mathbf{y}; \boldsymbol{\alpha}, \boldsymbol{\beta}) \star \Psi$.

6.3 Symbolic execution

The symbolic execution rules for our verifier are given in full in Appendix B. One such rule is:

LOOKUP

$$\frac{\begin{array}{c} \mathit{purify}(\Upsilon) \vdash_{SMT} e_A = (e'_A + o) \\ \Pi; \Gamma \triangleright \{x = (e[x\backslash x']) \ \star P \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\} \end{array}}{\Pi; \Gamma \triangleright \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, x := [e_A];\ C\, \{Q\}}\ x' \text{ fresh}$$

Most of these rules are similar in spirit, if not detail, to those found in [5]. In particular, we always reason about a command *followed by a "continuation" C*. This allows us to avoid introducing existential quantifiers; the previous values of variables can be represented with fresh variables such as x' in the rule above. It also demonstrates how the SMT solver is used to infer some pure facts (often for equational reasoning).

To show their soundness, we derive our symbolic execution rules from the core rules described in the previous section. As an example, Appendix C.1 gives such a derivation for (LOOKUP).

The rules which are intrinsically new in our work are those for the statements which make use of higher order store, namely $\mathsf{eval}\ [e_A](\mathbf{p})$ and $[e_A] := \mathcal{F}(\text{optparams})$. Here we just show a simplified version of the rule eval. The full version of this rule allows additional annotations to the statement, which help guide the proof. That version, and all the other rules, can be found in the Appendix.

$$\textsc{EvalUnguided}\quad \frac{\begin{array}{c}\Pi : \Upsilon \vdash_{\text{find-tr}} \exists \mathbf{y}.\, e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure} \\ \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\}\, [\mathbf{y} \backslash \mathbf{w}] \vdash_{\text{find-post}} \{\Upsilon \star R^{pure}[\mathbf{y}\backslash\mathbf{w}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\} \\ \Pi; \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i} \backslash \mathbf{v'_i}] \right\} C\, \{Q'\}\end{array}}{\Pi; \Gamma \triangleright \{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'});\ C\, \{Q'\}}\ \mathbf{v'_i}, \mathbf{w} \text{ fresh}$$

The assertion R^{pure} appearing in this rule contains (pure) information resulting from unfolding predicates or splitting lists during the search for an appropriate triple for the procedure stored in E. The need for including it in the precondition of the $\vdash_{\text{find-post}}$ judgment becomes clear when considering programs like in Figure 12. The first predicate definition represents a varying number of adjacent heap cells. The second predicate is a linked list segment containing code which expects two such \$Cell arguments. The number of cells for each argument can vary for each stored procedure in the list, but this typing information is contained within the abstract list $\%fs$.

```
recdef $Cells(ptr, n) :=
   n = 1 ⋆ ptr ↦ _ ∨ n = 2 ⋆ ptr ↦ _, _ ∨ n = 3 ⋆ ptr ↦ _, _, _;

recdef $CodeList(a, z; %fs) := a = z ⋆ %fs = ∅
∨  ∃n, %rest, t1, t2, %types.
     a ↦ ∀y, z. {$Cells(y, t1) ⋆ $Cells(z, t2)} · (y, z) {$Cells(y, t1) ⋆ $Cells(z, t2)}, n
     ⋆ %types = {(1, t1)} ∪ {(2, t2)}
     ⋆ $CodeList(n, z; %rest) ⋆ %fs = {(a, %types)} ∪ %rest;

proc main(x, y, f)
∀a, z, %fs, %types.
   pre : $Cells(x, 1) ⋆ $Cells(y, 2) ⋆ $CodeList(a, z; %fs)
      ⋆ (f, %types) ∈ %fs ⋆ %types = {(1, 1)} ∪ {(2, 2)};
   post : $Cells(x, 1) ⋆ $Cells(y, 2) ⋆ $CodeList(a, z; %fs);
{
   eval[f](x, y)
}
```

Fig. 12 Example demonstrating the need for R^{pure} in (EVAL)

Procedure *main* takes three arguments: the first and second point respectively to 1 and 2 adjacent heap cells; the third points to some code at address f that is

contained in the list beginning at address a. The final constraint in the precondition says that the code at f expects 1 cell and 2 cells in its first and second argument respectively. The body simply runs the code with the two arguments, which intuitively will succeed. We will now discuss why it is important to include extra pure information. The triple for f that is found by $\vdash_{\text{find-tr}}$ splitting the \$CodeList predicate is:

$$\forall y, z.\ \left\{\$\text{Cells}(y, t_1') \star \$\text{Cells}(z, t_2')\right\} \cdot (y, z) \left\{\$\text{Cells}(y, t_1') \star \$\text{Cells}(z, t_2')\right\}$$

where skolemization freshens the originally existentially quantified t_1, t_2 variables to fresh t_1', t_2'.

Then, in order to show the $\vdash_{\text{find-post}}$ hypothesis of (EvalUnguided) the rule (InferSpecForCall) from Appendix 6.4.3 requires to show that the current symbolic state (Υ) entails the precondition (P). Without including the extra pure part R^{pure}, the entailment thus is

$$\begin{array}{l} \$\text{Cells}(x, 1) \star \$\text{Cells}(y, 2) \star \$\text{CodeList}(a, z; \%fs) \star \\ (f, \%types) \in \%fs \star \%types = \{(1,1)\} \cup \{(2,2)\} \end{array} \quad \vdash \quad \begin{array}{l} \$\text{Cells}(x, t_1') \star \\ \$\text{Cells}(y, t_2') \end{array}$$

which does not hold because we do not remember anything about the values of t_1' and t_2' that had been revealed by the splitting. However, if we add the extra pure information that will have been exposed by the $\vdash_{\text{find-tr}}$ judgment using splitting the entailment is (for clearer presentation the non-crucial parts are omitted)

$$\begin{array}{l} \$\text{Cells}(x, 1) \star \$\text{Cells}(y, 2) \star \$\text{CodeList}(a, z; \%fs) \star \\ (f, \%types) \in \%fs \star \%types = \{(1,1)\} \cup \{(2,2)\} \star \\ \ldots \star \%types = \{(1, t_1')\} \cup \{(2, t_2')\} \end{array} \quad \vdash \quad \begin{array}{l} \$\text{Cells}(x, t_1') \star \\ \$\text{Cells}(y, t_2') \end{array}$$

which will hold because we can get the necessary equalities $t_1' = 1$ and $t_2' = 2$ through the assumption $\{(1, t_1')\} \cup \{(2, t_2') = \%types = \{(1,1)\} \cup \{(2,2)\}$. Note that it is legitimate to add R^{pure} temporarily to the current state Υ as we know by the $\vdash_{\text{find-tr}}$-assumption that $\Upsilon \Rightarrow true \star R^{pure}$ and thus $\Upsilon \Leftrightarrow \Upsilon \star R^{pure}$ (by axioms ($\star$-SplitPureRight) and ($\star$-SplitPureLeft)).

Simplification after each symbolic execution state

We can symbolically execute a procedure by symbolically executing the individual program statements. The top level procedure is `main()`. At the end of each program statement during symbolic execution, however, there is an optional "cleanup" phase designed to simplify the symbolic heap state. This is merely cosmetic with a goal of keeping the proof graphs more readable. There are three simplifications taking place:

1. Remove unused skolem variables
2. Remove redundant pure formula
3. Minimize $\mapsto$ for adjacent heap-cells

The first stage looks for an equality (between integers or sets) where at least one of the left or right-hand side is a skolem variable (identifiable by its ending

with a number). The skolem variable may then be substituted by the expression on the other side of the equality in the rest of the assertion, and the equality dropped.

$$\begin{array}{rl} \mathit{cleanup1}(\Phi) = \text{if } (v = e, \Theta) \text{ or } (e = v, \Theta) & \in \mathit{split}(\Phi) \\ \quad \text{and } \mathit{isSkolem}(v) & \\ \text{then } \Theta[v\backslash e] & \\ \text{else } \Theta & \end{array}$$

where $\mathit{split}(\Phi)$ returns a list of all ways of splitting out an atomic formula A from an assertion Φ, and returning a pair (A, Θ) where Θ is all the other formulae.

The second stage uses the SMT solver to remove any pure formulae which are implicit in the rest of the assertion. This is achieved by initially partitioning the assertion into two parts, ie. $\mathit{partition}(\Phi) = (\Phi^{spatial}, \Phi^{pure})$ where $\Phi^{spatial}$ is a spatial formula not containing any pure parts and Φ^{pure} is a pure formula such that $\Phi \equiv \Phi^{spatial} \star \Phi^{pure}$. Then, iterating through each pure atomic formula A in Φ^{pure}, we check whether A is implied by the new assertion being built, and drop A or add it to the new assertion accordingly.

$$\begin{array}{ll} \mathit{cleanup2}(\Phi) = & \mathit{cleanup2Aux}(\mathit{partition}(\Phi)) \\ \mathit{cleanup2Aux}(\Phi^{spatial}, \Phi^{pure}) = & \text{if } \Phi^{pure} \text{ is empty then } \Phi^{spatial} \\ & \text{else if } \Phi^{pure} \text{ matches syntactically } A \star \Theta^{pure} \\ & \quad \text{and } \mathit{purify}(\mathit{closure}(\Phi^{spatial})) \vdash_{SMT} A \\ & \quad \text{then } \mathit{cleanup2Aux}(\Phi^{spatial}, \Theta^{pure}) \\ & \quad \text{else } \mathit{cleanup2Aux}(\Phi^{spatial} \star A, \Theta^{pure}) \end{array}$$

The third stage looks for multiple points-to formula which, together, address adjacent heap cells. These formulae can then be condensed into a single points-to occurrence.

$$\begin{array}{rl} \mathit{cleanup3}(\Phi) = \text{if } \Phi \Leftrightarrow a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_n \star \Phi' & \\ \quad \text{and } \Phi' \Leftrightarrow a' \mapsto \mathscr{C}' \star \Theta & \\ \quad \text{and } \mathit{purify}(\mathit{closure}(\Phi)) \vdash_{SMT} a + n = a' & \\ \text{then } a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_n, \mathscr{C}' \star \Theta & \\ \text{else } \Phi & \end{array}$$

The cleanup is optional, set by a configuration flag. Whilst ordinarily it is desirable with no adverse effects, there is a cost in verification time (especially those stages using the SMT solver). Additionally, in the case of large assertions a pure formula may be removed that is syntactically identical to part of a future entailment goal. It is possible that the SMT solver reaches its timeout and verification will not succeed because the entailment is no longer trivial.

Soundness of Cleanup stages For all three stages, i.e. for $k = 1, 2, 3$, we can show that $\mathit{cleanupk}(\Phi) \Leftrightarrow \Phi$.

1. For *cleanup1* this follows from the fact that

$$\begin{array}{rl} h \in [\![\Phi[v\backslash e]]\!]_\rho & \Leftrightarrow h \in [\![\Phi]\!]_{\rho[v:=[\![e]\!]_\rho}} \\ & \Leftrightarrow h \in [\![\Phi]\!]_\rho \wedge \rho(v) = [\![e]\!]_\rho \\ & \Leftrightarrow h \in [\![\Phi]\!]_\rho \wedge \mathsf{emp} \in [\![v = e]\!]_\rho \\ & \Leftrightarrow h \in [\![\Phi \star v = e]\!]_\rho \end{array}$$

   for any heap h and environment ρ from which one can conclude that $\models \Phi \star v = e \Leftrightarrow \Phi[v\backslash e]$.

2. To show the soundness of *cleanup2* it suffices to consider the else case of the conditional and more precisely show that if Φ^{pure} matches syntactically $A \star \Theta^{pure}$ and $\mathit{purify}(\mathit{closure}(\Phi^{spatial})) \vdash_{SMT} A$ then $\Phi^{spatial} \star \Phi^{pure} \Leftrightarrow \Phi^{spatial} \star \Theta^{pure}$. From the second assumption it follows from soundness of SMT, Lemma 9 and (Closure) that $\Phi^{spatial} \Leftrightarrow \Phi^{spatial} \star A$. Thus by ($\star$-Monotonicity) we get that $\Phi^{spatial} \star \Phi^{pure} \Leftrightarrow \Phi^{spatial} \star A \star \Phi^{pure}$. By the first assumption and ($\star$-Associative) we get that $\Phi^{spatial} \star \Phi^{pure} \Leftrightarrow \Phi^{spatial} \star \Theta^{pure}$.
3. Soundness of *cleanup3* follows from the following considerations: From the third assumption of the conditional follows by the soundness of the SMT solver that $\mathit{purify}(\mathit{closure}(\Phi)) \Rightarrow a + n = a'$ so by Lemma 9 and (Closure) we get that $\Phi \Leftrightarrow \Phi \star a + n = a'$. Together with the other two assumptions in the condition we obtain that $\Phi \Leftrightarrow a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_n \star a + n \mapsto \mathscr{C}' \star \Theta$ from which the desired result follows by ($\mapsto$-Group).

6.4 Entailment proof search algorithms

6.4.1 Entailments between assertion disjuncts

The proof rules for this judgement are given in Figure 13, and their soundness is proved in Appendix D.

There are three stages in the proof search.

1. *Preparation.* First, blocks of consecutive heap cells, on both sides of $\vdash$, are broken up e.g. $x \mapsto a, 0$ is replaced by $x \mapsto a \star x{+}1 \mapsto 0$. Secondly, pure information which is implicit in the spatial parts on the left is made explicit using $\mathit{closure}(-)$, e.g. if the spatial parts are $x \mapsto \_ \star y \mapsto 0$ the pure constraints $x \neq 0$, $y \neq 0$ and $x \neq y$ are added. We need to do this because otherwise, once we start cancelling off spatial formulae we will lose this information.
2. *Cancelling spatial formulae.* This is the main part of the proof. We successively cancel spatial pieces from the left and right sides of $\vdash$, sometimes instantiating existentially quantified variables in the process. For example, the goal $\Upsilon \star x \mapsto 3 \vdash^{I} \exists u, \mathbf{v}.\ \Phi \star x \mapsto u \star \Theta$ is reduced to $\Upsilon \vdash^{I'} \exists \mathbf{v}.\ \Phi[u \backslash 3] \star \Theta$ by cancellation, where I will be $I'[u := 3]$. During this stage, calls to the prover for entailments between specifications may be necessary. For instance, when solving the goal $\Upsilon \star x \mapsto \forall \mathbf{a} \{P\} \cdot () \{Q\} \vdash^{I} \Phi \star x \mapsto \forall \mathbf{a} \left\{P'\right\} \cdot () \left\{Q'\right\} \star \Theta$ we can cancel the heap cells at x only if $\forall \mathbf{a} \{P\} \cdot () \{Q\} \vdash \forall \mathbf{a} \left\{P'\right\} \cdot () \left\{Q'\right\}$. which is, semantically speaking, $\forall \mathbf{a} \{P\}\, k() \{Q\} \Rightarrow \forall \mathbf{a} \left\{P'\right\} k() \left\{Q'\right\}$ for a fresh variable k. Sometimes different choices of instantiation for an existential variable lead to the cancellation of different spatial parts, so in general backtracking may be needed. The rules that give rise to backtracking are explicitly labelled as such. They are tried in the order in which they are listed. This means that applications of the rules which can backtrack are postponed as long as possible, which should be more efficient; if backtracking rules are used early, then other independent reduction steps may need to be repeated in many branches. Note that the rules are meant to match entailment problems modulo the order of the spatial conjuncts involved.

   Backtracking. As identified above, the four rules in this phase which backtrack are (InstMatchAddr), (InstMatchArg), (InstTripleVars) and (PureInst). In particular:

$$\textsc{InstUsingEq}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon[v\backslash e] \star \Theta}{\Phi \ \vdash^{I[v:=e]} \ \exists \mathbf{v}, v \,.\, \Upsilon \star v = e \star \Theta} \quad fv(e) \cap \mathbf{v}, v = \emptyset$$

$$\textsc{CancelPt1}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star \Theta}{\Phi \star e_A \mapsto \mathscr{C} \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star e' \mapsto \_ \star \Theta} \quad \begin{array}{l} fv(e') \cap \mathbf{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e' \end{array}$$

$$\textsc{CancelPt2}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star \Theta}{\Phi \star e_A \mapsto E \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star e' \mapsto E' \star \Theta} \quad \begin{array}{l} fv(e', E') \cap \mathbf{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e', \\ purify(\Phi) \vdash_{SMT} E = E' \end{array}$$

$$\textsc{CancelPred}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star \Theta}{\Phi \star P(e_1, \ldots, e_k) \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star P(e'_1, \ldots, e'_k) \star \Theta} \quad \begin{array}{l} fv(e'_1, \ldots, e'_k) \cap \mathbf{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} (e_1 = e'_1 \wedge \ldots \wedge e_k = e'_k) \end{array}$$

$$\textsc{CancelPtTriple}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star \Theta \qquad B_1 \vdash B_2}{\Phi \star e_A \mapsto B_1 \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon \star e' \mapsto B_2 \star \Theta} \quad \begin{array}{l} fv(e', B_2) \cap \mathbf{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e' \end{array}$$

$$\textsc{CancelPtInstContents}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon[v\backslash E] \star \Theta}{\Phi \star e_A \mapsto E \ \vdash^{I[v:=E]} \ \exists \mathbf{v}, v \,.\, \Upsilon \star e' \mapsto v \star \Theta} \quad \begin{array}{l} fv(e') \cap \mathbf{v}, v = \emptyset, \\ purify(\Phi) \vdash_{SMT} e_A = e' \end{array}$$

$$\textsc{InstMatchAddr}\quad \frac{\Phi \star e_A \mapsto \mathscr{C} \ \vdash^{I} \ \exists \mathbf{v} \,.\, (\Upsilon \star x \mapsto \mathscr{C}')[x\backslash e] \star \Theta}{\Phi \star e_A \mapsto \mathscr{C} \ \vdash^{I[x:=e_A]} \ \exists \mathbf{v}, x \,.\, \Upsilon \star x \mapsto \mathscr{C}' \star \Theta} \quad \textbf{backtracks}$$

$$\textsc{InstMatchArg}\quad \frac{\Phi \star P(e_1, \ldots, e_k) \ \vdash^{I} \ \exists \mathbf{v} \,.\, (\Upsilon \star P(e'_1, \ldots, e'_{r-1}, v, e'_{r+1}, \ldots, e'_k))[v\backslash e_r] \star \Theta}{\Phi \star P(e_1, \ldots, e_k) \ \vdash^{I[v:=e_r]} \ \exists \mathbf{v}, v \,.\, \Upsilon \star P(e'_1, \ldots, e'_{r-1}, v, e'_{r+1}, \ldots, e'_k) \star \Theta}$$

$$fv(e'_1, \ldots, e'_{r-1}) \cap \mathbf{v} = \emptyset, \quad \textbf{backtracks}$$

$$\textsc{InstTripleVars}\quad \frac{\Phi \vdash^{I} \exists \mathbf{v} \,.\, (\Psi \star e_A \mapsto B)[v\backslash w] \star \Theta}{\Phi \vdash^{I[v:=w]} \ \exists \mathbf{v}, v \,.\, \Psi \star e_A \mapsto B \star \Theta} \quad \begin{array}{l} w \in fv(\Phi), v \in fv(B) \\ \textbf{backtracks} \end{array}$$

$$\textsc{PureInst}\quad \frac{\Phi \ \vdash^{I} \ \exists \mathbf{v} \,.\, \Upsilon^{pure}[v\backslash w] \star \Phi}{\Phi \ \vdash^{I[v:=w]} \ \exists v, \mathbf{v} \,.\, \Upsilon^{pure} \star \Phi} \quad \begin{array}{l} w \in fv(\Phi), \\ \textbf{backtracks} \end{array}$$

$$\textsc{Pure}\quad \frac{purify(\Phi) \ \vdash_{SMT} \ \Upsilon^{pure}}{\Phi \ \vdash^{\emptyset} \ \Upsilon^{pure} \star \Phi}$$

$$\textsc{Identity}\quad \frac{}{\Phi \ \vdash^{\emptyset} \ \Phi}$$

Fig. 13 Rules for automatically proving entailments between assertion disjuncts.

- (InstMatchAddr). Here we should commit to the first available choice of cell from the RHS (that cell will have to be cancelled sometime, so there is no point trying other choices), but then backtrack trying different cells on the LHS.
- (InstMatchArg). Here we should commit to the first available choice of predicate use on the RHS (this will have to be cancelled sometime, so there is no point trying other choices), but then backtrack trying different predicate uses on the LHS.
- (InstTripleVars). Here we instantiate variables which appear inside a nested triple, but are actually quantified at the top level (outside the triple). We arbitrarily choose variables which appear in the LHS.
- (PureInst). Here we unintelligently guess instantiations for any remaining quantified variables that appear in a pure formula. It arbitrarily chooses a variable from the LHS.

Cutting. For efficiency reasons, some of the above rules can sometimes *cut* when they fail. Reminiscent of the cut operation ! in Prolog, this causes the search to abandon the current goal and return to the last point at which a backtrackable choice was made. Cutting is done when one of the rules can detect already that the current goal is not proveable.

For example, given the goal $\mathrm{P}(x) \vdash^I \mathrm{P}(x) \star y \mapsto \_$, the (CancelPt1) rule (which is tried before (CancelPred)) can already detect that the goal is unprovable because there is no way to cancel $y \mapsto \_$ from the right. Thus (CancelPred) cuts and this goal is abandoned, rather than wasting the time of using (CancelPred) to cancel $\mathrm{P}(x)$ from both sides only to get stuck later.

3. *Pure reasoning.* The cancellation rules are designed to reduce the goal to the form $\Upsilon \vdash^I \Phi \star \Theta$ where Φ is pure. We finish by sending the pure entailment problem $\mathit{purify}(\Upsilon) \vdash_{SMT} \Phi$ to an SMT solver (see rule (Pure)), and we take Υ as the inferred frame Θ, and the empty map for I.

6.4.2 Entailments between specifications

The four proof rules for the judgement $B_1 \vdash B_2$ are given in Figure 14, and their soundness is proved in Appendix E. The rules are applied in the order we present them.

The first three rules simplify the entailment problem by making the specification on right hand side simpler. The first rule, (RemoveForallRight), removes the (top-level) universal quantifiers, the second rule (DisjPre) deals with disjunctions in the precondition and the third rule (ExistsPre) removes any existential quantifiers in the precondition.

The final rule (TripleEnt) breaks down the checking of an entailment $B \vdash \{\Phi\} \cdot (\mathbf{t}) \{Q\}$ between specifications into two tasks. Intuitively, we first use $\vdash_{\text{find-post}}$ to try to compute a postcondition $\vee_{i=1}^{m} \exists \mathbf{v}_i . \Upsilon_i$ for the code with specification B when run in a state satisfying Φ. We then check whether $\vee_{i=1}^{m} \exists \mathbf{v}_i . \Upsilon_i$ implies the required postcondition Q, making sure no variable clashes can occur.

6.4.3 Inferring postconditions for invocations

The proof rules for this judgement are given in Figure 15, and their soundness is proved in Appendix F. The first rule we have for $\vdash_{\text{find-post}}$ instantiates $\forall$ quantifiers

$$\text{1 RemoveForallRight}\quad \frac{\forall \mathbf{y}.B \;\vdash\; B'[\mathbf{y}'\backslash\mathbf{w}]}{\forall \mathbf{y}.B \;\vdash\; \forall \mathbf{y}'.B'}\ \mathbf{w}\text{ fresh}$$

$$\text{2 DisjPre}\quad \frac{\bigwedge_{i=1}^{n} \forall \mathbf{y}.B \;\vdash\; \{\exists \mathbf{v}_i.\Phi_i\}\cdot(\mathbf{t})\,\{Q\}}{\forall \mathbf{y}.B \;\vdash\; \{\exists \mathbf{v}_1.\Phi_1 \vee \ldots \vee \exists \mathbf{v}_n.\Phi_n\}\cdot(\mathbf{t})\,\{Q\}}$$

$$\text{3 ExistsPre}\quad \frac{\forall \mathbf{y}.B \;\vdash\; \{\Phi[\mathbf{v}\backslash\mathbf{a}]\}\cdot(\mathbf{t})\,\{Q\}}{\forall \mathbf{y}.B \;\vdash\; \{\exists \mathbf{v}.\Phi\}\cdot(\mathbf{t})\,\{Q\}}\ \mathbf{a}\text{ fresh}$$

$$\text{4 TripleEnt}\quad \frac{B \;\vdash_{\text{find-post}}\; \{\Phi\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon_i\right\} \qquad \bigwedge_{i=1}^{m} \Upsilon_i[\mathbf{v}_i\backslash\mathbf{a}_i] \;\vdash^{I_i}\; \exists \mathbf{b}_{j_i}.(\Upsilon'_{j_i}[\mathbf{w}_{j_i}\backslash\mathbf{b}_{j_i}]) \star \Theta_i}{B \;\vdash\; \{\Phi\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m'} \exists \mathbf{w}_i.\Upsilon'_i\right\}}$$

1. $j_1, \ldots, j_m \in \{1, \ldots, m'\}$,
2. $\mathbf{a}_1, \ldots, \mathbf{a}_m$ all chosen fresh,
3. $\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_{m'}}$ all chosen fresh
4. $\Theta_1, \ldots, \Theta_m$ pure

Fig. 14 Rules for automatically proving entailments between specifications.

$$\text{InstParam}\quad \frac{(\forall \mathbf{a}\,.\,\{P\}\cdot(\mathbf{t_1}, y, \mathbf{t_2})\,\{Q\})\,[y\backslash t] \;\vdash_{\text{find-post}}\; \{\Phi\}\cdot(\mathbf{t_1}, t, \mathbf{t_2})\,\{Q'\}}{\forall \mathbf{a}, y\,.\,\{P\}\cdot(\mathbf{t_1}, y, \mathbf{t_2})\,\{Q\} \;\vdash_{\text{find-post}}\; \{\Phi\}\cdot(\mathbf{t_1}, t, \mathbf{t_2})\,\{Q'\}}\quad \mathbf{t_1} \cap (\mathbf{a} \cup \{y\}) = \emptyset$$

$$\text{InferSpecForCall}\quad \frac{\Phi \;\vdash^{I}\; \exists \mathbf{u}_k, \mathbf{a}.\Upsilon_k \star \Theta}{\forall \mathbf{a}.\left\{\bigvee_{i=1}^{n} \exists \mathbf{u}_i.\Upsilon_i\right\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right\} \;\vdash_{\text{find-post}}\; \{\Phi\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m} (\exists \mathbf{v}_i.\Upsilon'_i[\mathbf{a}\backslash I(\mathbf{a})] \star \Theta)\right\}}$$

1. $\mathbf{t} \cap \mathbf{a} = \emptyset$,
2. $fv(\Phi) \cap \mathbf{u}_k, \mathbf{a} = \emptyset$,
3. for each $i \in \{1, \ldots, m\}$ we have $\mathbf{v}_i \cap \mathbf{a} = \emptyset$
4. for each $i \in \{1, \ldots, m\}$, no formula in $I(\mathbf{a})$ contains a variable from $\mathbf{v}_i$
5. $k \in \{1, \ldots, n\}$
6. no formula in $I(\mathbf{u}_k)$ contains a variable from $\mathbf{a}$
7. $\mathbf{u}_k \cap \mathbf{a} = \emptyset$

backtracks

Fig. 15 Rules for automatically inferring postconditions for procedure invocations.

on the left hand side so that the parameters in the two specifications match. (The side condition here makes sure that we instantiate the *leftmost* quantified parameter, for the sake of determinism.)

The main rule for $\vdash_{\text{find-post}}$ is (InferSpecForCall). Underlying the rule is a combination of $\forall$-instantiation, and the (ShallowFrameProcedures) and (ConsequenceProcedures) axioms.

Note that after (InstParam) has been used to make the parameters match, some bound variables may need to be renamed to fresh variables to allow the (InferSpecForCall) rule to be used. We do not go into detail about these renamings.

$$\textsc{Find}\quad \frac{P \vdash_{SMT} e'_A = e_A + o}{\Pi : P \star e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, B, \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \vdash^{\varepsilon}_{\text{find-tr}} e'_A \mapsto B}$$

$$\textsc{FindUnfold}\quad \frac{\begin{array}{c} \mathit{purify}(\mathit{closure}(\Psi_i[\mathbf{v}_i\backslash\mathbf{w}_i][\mathbf{v}\backslash\mathbf{E}] \star P)) \vdash_{SMT} \mathit{false} \text{ for all } i \neq k \\ \Pi, \mathrm{X}(\mathbf{v}) \Leftrightarrow \exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n : \Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}] \star P \vdash^{\varepsilon}_{\text{find-tr}} \exists \mathbf{a}.\, e_A \mapsto B \star R^{pure} \end{array}}{\Pi, \mathrm{X}(\mathbf{v}) \Leftrightarrow \exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n : \mathrm{X}(\mathbf{E}) \star P \vdash^{\varepsilon}_{\text{find-tr}} \begin{array}{l} \exists \mathbf{a}, \mathbf{w_k}.\, e_A \mapsto B \\ \star\ \mathit{purify}(\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{y}\backslash\mathbf{E}]) \star R^{pure} \end{array}}$$

$$\mathbf{w}_1, \ldots, \mathbf{w}_n \text{ fresh}$$

$$\textsc{FindSplit}\quad \frac{\begin{array}{c} \mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := Q \in \mathrm{LsegDefn}(s, n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_K], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \\ P \vdash^{I} \exists u_1, \ldots, u_k\ .\ (u_1, \ldots, u_k) \in \hat{e}_\gamma \star \Theta \\ S = \begin{pmatrix} \mathrm{L}(\hat{e}, s, \alpha) \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_K, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, \hat{e}', \beta) \\ \star\ \hat{e}_\gamma = \alpha \cup \{(I(u_1), \ldots, I(u_k))\} \cup \beta \\ \star\ E_1 = I(u_1) \star \cdots \star E_k = I(u_k) \end{pmatrix} \\ \Pi, \mathrm{L}(\mathbf{x}) \Leftrightarrow Q : S[s, n, \mathbf{v}\backslash\mathbf{w}] \star P \vdash^{\varepsilon}_{\text{find-tr}} e_A \mapsto B(\cdot) \star\ R^{pure} \end{array}}{\Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow Q : \mathrm{L}(\hat{e}, \hat{e}', \hat{e}_\gamma) \star P \vdash^{\varepsilon}_{\text{find-tr}} e_A \mapsto B(\cdot) \star\ \mathit{purify}(S[s, n, \mathbf{v}\backslash\mathbf{w}]) \star R^{pure}}$$

$$\alpha, \beta, \mathbf{w} \text{ fresh},\ u_1, \ldots, u_k \text{ fresh}$$

Fig. 16 Rules for finding specifications inside a symbolic state

6.4.4 Finding specifications inside a symbolic state

To be able to symbolically execute an $\mathsf{eval}\ [e](\mathbf{p})$ statement, we need to first find in our symbolic heap a cell $e \mapsto B$; we can then use the specification B to reason about the invocation. We use $\vdash_{\text{find-tr}}$ for finding such specifications. The proof rules for this judgement are given in Figure 16.

When the required cell $e \mapsto B$ is explicitly present in the symbolic heap, finding it is easy; the (Find) rule covers these cases. In practise, however, the specification is often "packaged up" inside a user-defined predicate, and requiring such predicates to be explicitly unfolded or split to reveal the specification would be extremely inconvenient. Therefore, our prover for $\vdash_{\text{find-tr}}$ does a limited amount of such unfolding and splitting automatically, as described by the (FindUnfold) and (FindSplit) rules. The exact limitation is determined by a constant that prescribes the "depth" of the nested unfoldings which is kept small to reduce the search space. An extension to address this issue will be presented in Section 6.6.

6.5 An example of proving entailments between specifications

We mentioned in Section 3 that a key step in the verification of our example program is proving that the strong specification for the list library entails the weaker variation (see the entailment (2)). We now discuss this point in more detail, emphasising how the entailment prover for assertions and the prover for specifications are mutually recursive.

Before the call to the *useFib* procedure in *main*, the $\$ListLibraryStrong$ predicate is unfolded, and folded up into $\$ListLibraryWeak$. This essentially means proving (2) which is an entailment between assertion disjuncts.

The proof proceeds by cancelling out the atomic formulae, which in this case means using (CancelPtTriple) for each of the four library procedures. This is where the entailment prover for specifications is needed: the premise of this rule requires that each strong specification entails the respective weak variation.

This entailment is checked by the (TripleEnt) rule, which has two premises. The first uses the judgement $\vdash_{\text{find-post}}$ – with (InferSpecForCall) – which will check that the weak precondition entails the strong precondition, with some inferred frame left over (in this case the frame is trivial). For the second premise it is required to prove that the strong postcondition (together with the frame) entails the weak one. Using again the entailment prover for assertion disjuncts, one obtains the following:

$$\$AssocListH(al, \{key\} \cup \kappa) \vdash^{[\kappa' \mapsto \{key\} \cup \kappa]} \exists \kappa'.\ \$AssocListH(al, \kappa')$$

The detailed steps of the reasoning are reproduced in Appendix I.

6.6 Advanced hints

In Section 2.3.2, one saw hints for the prover in the form of instantiation hints, for quantified variables, and ghost-statements, used during the symbolic execution. For some specific example programs, a need arose for the provision of additional hints. These advanced hints allow the user to gain explicit control over the automated built-in entailment proof search algorithms to perform complex logical reasoning the automated algorithms are unable to perform on their own (due to restricted power of the SMT solver and limited proof search). We distinguish two categories:

- for eval: user-guidance for finding the right triple by using ghost unfolds and splits and using a lemma to show the entailment between the stored procedure's precondition and current state.
- for storing procedures: relaxing the condition that the stored procedure must meet the specification prescribed by the current state and allowing a user-guided entailment proof between established and required triple in terms of ghost folds in pre- and postcondition.

6.6.1 Predicate folding for finding triples and lemma application for entailment

$$
\begin{array}{rcl}
\text{atomicst} & ::= & \ldots \\
 & & \mid \mathsf{eval}\ [\text{aexp}](x^*)\ \ \text{inst-hints? lookup? lemma-app?} \\
 & & \mid \ldots \\
\text{lookup} & ::= & \mathsf{before}\ \text{ghostopen}^* \\
\text{lemma-app} & ::= & \mathsf{after}\ \mathcal{L}(t^*)
\end{array}
$$

The eval statement has been enriched with two hooks into the execution of the eval rule. The "lookup" hint that may be provided to eval $[a](\ldots)$ tells the prover where to find the triple at address a in the cases where it is (deeply) hidden inside predicate instances. It is a list of ghost unfold/split statements. As mentioned in Section 6.4.4, without the hints if the triple is not explicitly visible in the symbolic heap state our verifier will only do a limited amount of automatic unfolds/splits which, however, are carried out in an unintelligent and inefficient way when there are a large number of predicate instances in the current symbolic heap. Accordingly, the $\vdash_{\text{find-tr}}$ judgement presented earlier is extended to allow the provision of a sequence of ghost statements $\mathbf{G}$: $\Pi : \Upsilon \vdash^{\mathbf{G}}_{\text{find-tr}} e \mapsto B \star R^{pure}$. This gives the user some control over the search for matching nested triples. The two extra rules which make use of the $\mathbf{G}$, (FindGuidedUnfold) and (FindGuidedSplit), can be found in Appendix G.

The "lemma-app" hint provides a hook to apply a lemma to the current symbolic state, before the stored procedure is evaluated. First we have to explain what we mean by a lemma as there are no extra syntactic lemma declarations. We rather follow VeriFast [30, 29] and make use of the following observation:

$$\{P\}\,\mathsf{skip}\,\{Q\} \iff P \Rightarrow Q$$

Thus, an entailment lemma of the form $P \Rightarrow Q$ can be declared simply as a procedure with precondition P, postcondition Q and body skip. This is desirable because the lemma will be automatically proved (with the additional help of ghost statements maybe). If the lemma is an abstract procedure, then the obligation to prove it falls to the user to undertake elsewhere. This feature is useful when the SMT solver is unable to complete the proof.

6.6.2 Predicate folding for triple entailment

atomicst	::=	$\ldots \mid [\text{aexp}] := \mathcal{F}(\text{optparam}^*)$
		deepframe? storecode-pre? storecode-post?
storecode-pre	::=	pre ghost-fold$^+$
storecode-post	::=	post ghost-fold$^+$

The next class of hint concerns the statement for storing code. Consider the example in Figure 17.

This example uses two predicates, $List representing a list of procedures, and $A representing the pre/postcondition of the procedures which appear in the list. To make the code-list definition easily reusable, the specification has been parameterized by the predicate $A. Thus, for different examples the same list definition can be used by providing different definitions of the $A predicate.

In procedure $\mathcal{F}$ we have a pointer l to a list. The line $[f] := \mathcal{G}(\_)$ adds procedure $\mathcal{G}$ to the head of the list, resulting in a new list starting at f. But $\mathcal{G}$'s footprint is smaller than $A, which is allowable because we can use the deepframe annotation to frame on the additional constraint(s). During the symbolic execution of $\mathcal{F}$, it is then necessary to prove the entailment

$$f \mapsto \forall r.\ \{r \mapsto \_ \star c \mapsto \_\} \cdot (r)\,\{r \mapsto \_ \star c \mapsto \_\} \vdash f \mapsto \forall x.\ \{\$\mathrm{A}(x)\} \cdot (x)\,\{\$\mathrm{A}(x)\}$$

```
const c;
recdef $A(a) := a ↦ _ ⋆ c ↦ _;
recdef $List(a; %A) := a = 0 ⋆ %A = ∅
      ∨ ∃n, %B. a ↦ ∀x. {$A(x)} · (x) {$A(x)}, n ⋆ $List(n; %B) ⋆ %A = %B ∪ {a};
```

```
proc F(l)
∀ %L.
pre : ∃hd.  l ↦ hd ⋆ $List(hd; %L);
post : ∃hd, %L2.  l ↦ hd ⋆ $List(hd; %L2) ⋆ %L ⊆ %L2;
{
  locals head, f;
  head := [l];
  f := new 0, head;
  [f] := G(_) ‘ deepframe
            c ↦ _ pre fold $A(?) post fold $A(?) ’
  ghost ‘fold $List(f; ?)’
  [l] := f
}
```

```
proc G(r)
pre : r ↦ _;
post : r ↦ _;
{ skip }
```

Fig. 17 Example demonstrating additional hints for the store-code statement.

It is clear that the nested pre- and postcondition are precisely $\$\mathrm{A}(r)$, however the proof algorithm for entailment of triples does not do automatic folds or unfolds (since they are expected to be done in ghost statements in the procedure body only) and thus needs to be told that this is the case. The two extra pre and post annotations to the store-code statement hint serve this purpose and can give the necessary fold instruction. A demonstration of the use of such hints in a more realistic setting can be seen in the verification of the reflective visitor pattern [28].

To establish the extra annotations the entailment judgement $\vdash^{I}$ needs to be able to use the predicate context, and take a fold command giving rise to the following enriched form of judgement:

$$\Pi : \Phi \vdash^{I}_{G} \exists \mathbf{v}.\Upsilon \star \Theta$$

The enriched judgement is supported by two extra entailment rules, which are used by a new (StoreCodeGuided) rule for storing procedures. These rules are given in Figure 18, and their soundness is proved in appendices C.5 and D. Note that this rule can only be applied to cases where the triple being manipulated does *not* contain a disjunction in its pre- or postcondition. This has been sufficient for our examples, however the rule could be generalised by iterated uses of the $\vdash^{\emptyset}_{G}$ in the premise.

7 Related work

The verification algorithms presented and proved correct in this paper have been implemented in a tool, Crowfoot, that can be considered as extending Smallfoot

$$\textsc{StoreCodeGuided}$$

$$\frac{\begin{array}{c}\Pi : ((\Phi \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \;\vdash^{\emptyset}_{G^{\mathrm{pre}}}\; X(\mathbf{d}) \star \Phi' \\ \Pi : ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}] \star \mathsf{emp} \;\dashv\vdash^{\emptyset}_{G^{\mathrm{post}}}\; Y(\mathbf{e}) \star \Theta' \\ S = \left(\begin{array}{c} \{\, \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{d})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \,\} \\ \forall \mathbf{p}|_U, \mathbf{x}. \qquad \cdot(\mathbf{p}|_U) \\ \{\, \exists \mathbf{w}, \mathbf{a'}.\ (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}] \,\} \end{array}\right)[\mathbf{p}|_{I\backslash U}\backslash\mathbf{t}|_{I\backslash U}] \\ \mathit{purify}(\Upsilon) \vdash_{SMT} E = E' + o \\ \Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\, \mathcal{F}(\mathbf{p})\, \{\exists \mathbf{w}.\Theta\} \triangleright \{\, \Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n \,\}\; C\; \{Q'\} \end{array}}{\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\, \mathcal{F}(\mathbf{p})\, \{\exists \mathbf{w}.\Theta\} \triangleright \{\, \Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n \,\} \quad \begin{array}{l} [E] := \mathcal{F}(\mathbf{t}) \\ \text{`}\mathsf{deepframe}\ \exists \mathbf{a}.\ \Upsilon' \\ \mathsf{pre}\ G^{\mathrm{pre}} \\ \mathsf{post}\ G^{\mathrm{post}}\text{'};\ C \end{array} \quad \{\, Q' \,\}}$$

where $|\mathbf{t}| = |\mathbf{p}|$ and t_i either value expression a_i or _;
$\mathbf{x} = \mathit{fv}(S\text{-precondition}, S\text{-postcondition}) - \mathbf{p}$; $\mathbf{v'}, \mathbf{w'}, \mathbf{a'}, \mathbf{v''}, \mathbf{w''}$ fresh
$\mathbf{p} = (p_i)_{i \in I}$; $U = \{i \in I \mid t_i = \_\}$ $\mathbf{p}|_X = (p_i)_{i \in I \cap X}$

$$\textsc{FoldPredRight}$$

$$\frac{(\mathrm{X}(\mathbf{v}) \Leftrightarrow (\exists \mathbf{v}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \mathbf{v}_n.\ \Upsilon_n)) \;\in\; \Pi \qquad \bigwedge_{1 \le i \le n} (\Upsilon_i[\mathbf{v}\backslash\mathbf{e}][\mathbf{v_i}\backslash\mathbf{d_i}] \star \Theta \;\vdash^{\emptyset}\; \Phi \star \mathsf{emp})}{\Pi : X(\mathbf{e}) \star \Theta \;\vdash^{\emptyset}_{G}\; \Phi \star \mathsf{emp}} \quad \begin{array}{l} \mathbf{d_1} \ldots \mathbf{d_n} \text{ fresh} \\ G = \mathsf{fold}\ X(\_)\ \_ \end{array}$$

$$\textsc{FoldPredLeft}$$

$$\frac{\mathrm{X}(\mathbf{v}) \Leftrightarrow (\exists \mathbf{v}_1, \mathbf{b}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \mathbf{v}_n, \mathbf{b}_n.\ \Upsilon_n) \;\in\; \Pi \qquad \Phi \;\vdash^{I}\; \exists \mathbf{w}, \mathbf{c}.\ \Upsilon_i[\mathbf{v}\backslash\mathbf{q}][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}] \star \Theta}{\Pi : \Phi \;\vdash^{\emptyset}_{G}\; X(\mathbf{e}) \star \Theta}$$

$G = \mathsf{fold}\ X(\mathbf{p})\ \mathbf{b} = \mathbf{y}$,
$\mathbf{b_i} \subseteq \mathbf{b}$, $\mathbf{b} - \mathbf{b_i} \cap (\mathit{fv}(\Upsilon_i) \cup \mathbf{p}) = \emptyset$
$1 \le i \le n$, $\mathbf{c}, \mathbf{w}$ fresh,
each p_i is an expression or ?,
each q_i is $\begin{cases} c_i & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}$,
each e_i is $\begin{cases} I(c_i) & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}$

Fig. 18 Additional symbolic execution and entailment rules for store-code hints

[5] (though Crowfoot was written from scratch) by allowing (partially applicable) procedures to be stored on the heap and to be invoked from the heap. The assertion language uses nested triples to specify stored procedures and recursively defined assertions to deal with recursion through the store. An SMT solver is invoked to deal with pure assertions and therefore Crowfoot can be used to prove more than just memory safety (see [13,28], and the example in this paper where *fib* is proved to compute Fibonacci numbers).

Nowadays, there exist numerous logics and verification systems that use separation logic and some of them are Coq extensions. In this section we will *only* focus on the ones that support *higher-order store*. A detailed comparison of effectiveness and user friendliness between automatic tactics in Coq-based verifiers (like CFML, Ynot or Bedrock) and the (semi-)automation provided by dedicated tools (like jStar, VeriFast and Crowfoot), respectively, is difficult and beyond the scope of this article. It is subject to further research. One point is obvious though, all

Coq extensions automatically have access to a second order assertion logic which the stand-alone systems do not have or have only in a weak sense[12].

7.1 Stand-alone systems following in the footsteps of Smallfoot

VeriFast The system most closely related to Crowfoot is the VeriFast [30,29] tool, also based on symbolic execution with separation logic. VeriFast supports a C-like language (and also Java) and supports C-style function pointers. Functions in the C-like language live in an immutable memory and can be pointed to but not updated, whereas Crowfoot's programming language stores procedures in dynamic, mutable memory. However these setups seem to have a similar character.

A key difference is that while Crowfoot uses nested triples to express requirements for procedure pointers, VeriFast expresses such requirements via *function types* with which the C type system is extended. A function type declaration associates a pre- and postcondition with the function type; the declared type can have extra arguments to simulate nested triples which can contain free variables. These can be recursive since for every function type F there is a predicate 'is_F(_)' which states that (the function pointed to by) its first argument satisfies the "contract" for function type F (possibly with additional arguments).

The presented tool also offers some features which VeriFast does not, such as partial application of which our example makes essential use in *useFib* when loading the memoiser *mem*. Another important feature to support stored procedures is entailment between Hoare triples which is automated in our verifier and needed in our example, as explained in Section 6.5. VeriFast does not support such proofs (which in that system would be proofs of entailments of shape $\text{is\_}F(\_) \Rightarrow \text{is\_}G(\_)$), even manual ones, whereas Crowfoot finds them automatically. Crowfoot supports annotations for deep frame rule application and allows extensions of (recursive) predicates, thus allowing elegant use of deep framing on recursively defined specifications (cf. definition of $\$S$ in our example in Figure 4). In VeriFast one can simulate the effect of the deep frame rule by using (second order) function types which take as argument a predicate representing the deeply framed invariant. However, this means one must write all specifications that can appear for stored procedures *a priori* in that style.

On the other hand, VeriFast offers features that Crowfoot does not, such as concurrency, termination checking and the use of more types (such as mathematical lists and functions on them) in the assertions. VeriFast's support for second order logic is useful for specifying and reasoning about higher-order and polymorphic functions.

7.2 Shallow embeddings in Coq

There exist a number of verification systems for higher-order store based on separation logic that have been developed in Coq: XCAP [33] a logic for assembly code, Bedrock [18] which provides significant automation for the XCAP logic, GCAP

[12] In Crowfoot we can quantify over predicates but only on the outermost level ie. we cannot substitute predicate variables.

[10], an extension of XCAP for self-modifying code, CFML [12], a tool for verifying ML functions including references using characteristic functions, Ynot [32] implementing Hoare Type Theory as a Coq library with some efficient Coq tactics added in [19].

XCAP/GCAP Both systems have been developed to prove correctness of assembly programs. In contrast to the Crowfoot logic, which supports a high-level C-like language, the CAP (certified assembly programs) approach is *intensional*. This means that assembly code is syntactically saved as first-order data in the store, so that it can be manipulated and (partially) overwritten. The Crowfoot language treats code *extensionally*, and therefore code can only be stored or invoked. Since the stored procedures can have arguments, their invocation allows a form of code configuration through their actual parameters. Code manipulation is limited by those features. This resembles the extended CAP approach, XCAP [33], that uses embedded code pointers. Modularity is achieved by adding new syntax for propositions, `cptr(f,a)`, expressing that precondition `a` holds for the code at label `f`. The interpretation of this new assertion syntax is only ever done when the entire program and its code labels are known. When proving correctness locally one only uses implications between assertions, a kind of relative correctness. In Crowfoot, modularity is ensured by using nested triples and procedures with arguments can be stored on the heap as a high-level language is used.

The "general" CAP approach, GCAP, drops "the assumption that code memory is fixed and immutable" [10] to support code generation and manipulation. The framework is platform independent using a General Target Machine that can be instantiated easily by specifying the intended operational semantics. The intensional descriptions of code blocks themselves have to be added to the assertions in program specifications. This approach is again based on the intensional representation of code and provides great flexibility. During verification, one has to keep carrying around the code block descriptions used for updates. It is pointed out that this "does not compromise modularity, since the code is already present" [10]. Correspondingly, *code specifications* map code blocks to assertions.

By contrast, Crowfoot code is always described by Hoare triples, the intensional representation is forgotten. The flexibility of updating code (and its specification) is ensured by the feature of nested Hoare triples that can express what specification a code pointer is supposed to satisfy in another procedure's pre- or postcondition.

Both program logics have been proved sound. While inductive techniques traditionally used with operational semantics are employed for XCAP/GCAP, the Crowfoot soundness proof makes use of a hybrid semantics where a standard operational programming language semantics is in use but a rather sophisticated interpretation of assertions using Kripke semantics combined with step-indexing.

Bedrock is a Coq framework that supports mostly-automated XCAP proofs. This provides semi-automated proofs with hints and achieves something very similar to what Crowfoot offers. In both systems automation makes use of simplification of assertions, automatic unfolding of predicates where possible, and cancellation of spatial conjuncts until only a pure fact remains. In Bedrock this is all implemented with the help of Coq's `Ltac` tactical language, a domain-specific language for proof-search. Bedrock is platform-independent and supports the XCAP logic, ie. embedded code pointers but not the more involved GCAP logics for self-modification.

CFML supports the verification of (a subset of) Caml programs using the Coq proof assistant. Caml programs are higher-order functional programs with general references. Therefore CFML supports higher-order store. CFML works slightly differently from other verifiers in the sense that it builds a characteristic formula for the given Caml program that is then used in the proof. This approach follows ideas as presented in [26]. The characteristic formula of a program term is a proposition expressed in higher-order logic that describes the semantics of Hoare triple for this term (where pre- and post-condition are given as parameters) without referring to the program syntax itself. It describes the axiomatic semantics of the program term in purely logical form. The characteristic formula of a given program applied to the pre- and post-condition of a given specification can then be proved in Coq using logical means provided that separation logic has been embedded shallowly to describe heaps [13] and program values are reflected in the logic (which is non trivial for functions). A set of Coq tactics have been developed and where hints are necessary the proof has to be carried out interactively. This approach is only limited by the expressiveness of the theorem prover used (here Coq) and a relative completeness theorem has been shown which is one particular bonus of using characteristic formulae.

Ynot [32] is an axiomatic extension to Coq's type theory in which Hoare triples ("Hoare types") can be used as the types for side-effecting commands; these Hoare types can be nested. Mixing Hoare triples with types permits the inclusion of Hoare specifications in abstract data types and thus elegant modular specification of imperative higher-order programs. Hoare Type Theory does not support recursion through the store set up on the fly as discussed in [32][Section 9] but admits predefined knots in the store with the help of recursive specifications. In Crowfoot we do not have this restriction but we also do not have a built-in type theory. Types in Crowfoot must be explicitly described as predicates. Improved tactics for Hoare Type Theory in Coq based on `Ltac` have been implemented in [19].

8 Crowfoot Usability Report

An interactive version of Crowfoot can be used online [1], which includes the example of this paper. Additionally, our earlier work describes several other applications of the logic:

1. verification of runtime code updates [13]
2. verification of a program that evaluates expressions in a binary tree [16]
3. verification of an instance of the reflective visitor pattern [28]
4. verification of a higher-order list search algorithm (online example)
5. verification of an updatable login server (online example)
6. inspired by media players, verification of a program that allows user triggered loading/unloading of plugins (online example)

We'll now briefly discuss our practical experiences.

The Crowfoot tool was developed incrementally, adding new language and verification features, and improving automation as warranted by the increasing complexity of examples.

[13] The Conjunction rule is not needed here, nor is it in Crowfoot.

In terms of automation, the provision of hints are the main manual burden. Because the verification is not interactive, a degree of prescience is required in order to ensure required predicate fold/unfold instructions take place at the correct point in the proof tree. This should be fairly intuitive for the same human verifier who constructed the definitions of predicates. A difficulty arises however when there is a choice for unfolding/folding a predicate, requiring predicate arguments to be provided in the annotations. In the event that these variables have been skolemized, it will not be easy to know the actual variable name until attempting to verify the program.

A problem that is specific to the higher-order store class of programs, particularly recursive ones, is that an eval command may need to unfold predicates in order to expose the specification of the code. Whilst the tool will automatically start unfolding predicates if the contents of the address is not accessible, this blind searching will slow verification time in cases where there are many predicates instances. The alternative is to provide the "lookup" hints as discussed in Section 6.6.1.

These drawbacks can be overcome to some extent by the graphical visual output that Crowfoot produces. This output represents the proof tree and, if the automatic proof failed, it is highlighted where in the tree the proof got stuck. In most cases the problem is trivially identified – be it from a missing predicate unfold or an incompatible specification.

One of the limitations that can arise during checking of entailments, particularly when assertions contain complex set expressions, is that the SMT-solver is unable to decide pure assertions before a reasonable timeout. It is not the case that all entailments sent to the solver are expected to be correct, because the existential variable instantiation can choose the wrong variables on the first attempt. If the SMT solver timeout was globally extended, then the total verification time would be vastly increased. Therefore it can be desirable to have *variable* timeouts. For instance, the symbolic execution of a procedure call may require showing the entailment of a complex pre-condition and the SMT solver may need 200ms. However a simple dereferencing operation may succeed in under 50ms.

An approach that is often necessary for complex inductive proofs or pure entailments that stretch the SMT solvers' capabilities is to use lemmas in the form of auxiliary procedure calls. This approach is also used by VeriFast. As discussed in Section 6.6.1, if a procedure is defined using only ghost statements or skip, then an implication has been proved between the pre- and post-condition. The use of lemmas has been explored in more detail in the extensions presented in [27].

The verification of more complex programs, where the symbolic state contains a large number of distinct variables, can have a significant cost in terms of verification time. This is because of the simple heuristic decisions that are made by Crowfoot for instantiating existentially quantified variables, and the possibility of backtracking. The chance of the correct variable being chosen will be rare if the more intelligent instantiation rules do not apply.

There is scope for improving the level of automation, and also the efficiency of the verification. Predicates could be intelligently unfolded/folded by the system. For instance when a post-condition includes a predicate instance that is not present in the state after symbolic execution, the prover could try to fold. Additionally, if a program statement uses a variable that appears in the arguments of a predicate

instance, and the symbolic execution gets stuck, that predicate instance could be unfolded.

The available configurations of hints cover most of the cases where the entailment prover may need to "guess" instantiations for existentially quantified variables. The exception is at the point when the symbolic execution of a procedure has completed and the symbolic heap is checked against the post-condition. It would be trivial to extend the current system of hints to allow the guiding of the instantiation of existential variables in the post-condition.

9 Future work

The following extensions would permit the verification of more examples. Firstly, as the *antiframe rule* is consistent with the logic used here (as proved in [42]), annotations similar to those for the deep frame rule could be implemented to allow hiding of invariants in "antiframe style". Although we do not need it for deep framing like VeriFast does, second order logic would support the specification of parametric procedures. A minor but useful extension is to allow proper functions with result values. The already quite general Lemma 7 can be generalised to support mutually recursive definitions and to allow deep framing onto abstract (universally quantified) predicates. To achieve this, for each abstract predicate P one can also define another abstract predicate $\mathrm{P} \otimes \Psi$ with a corresponding equivalence $\forall \mathbf{x}, \boldsymbol{\alpha}.\ \mathrm{P}(\mathbf{x}; \boldsymbol{\alpha}) \otimes \Psi \Leftrightarrow \mathrm{P} \otimes \Psi(\mathbf{x}; \boldsymbol{\alpha})$. There is scope for improving the efficiency of the automate procedure regarding disjunctions, see e.g. [22]. Some support for the "magic wand" could simplify the verification of programs that manipulate more complex data structures like e.g. trees. Some suggestions and ideas have been proposed recently in [9,39,31]. A semantic checker could be added that verifies that a recursive predicate declaration is actually an instance of a pattern that guarantees the predicate's existence. Such a pattern is described in [16].

Finally, there are some possible enhancements relating to hints. Firstly, it is likely that many fold/unfold annotations can be discovered automatically, as done in [17]. Next, in addition to the advanced hints in Section 6.6 that were introduced to support verification of reflective programs, we plan to investigate what further extensions are required to support reasoning about more features of reflection. Lastly, looking at hints in a more general sense, the need for advanced hints discussed in Section 6.6 suggests that a more conceptual approach regarding user interaction (ghost statements) in entailment proofs might be beneficial. For instance, one could provide "hooks" in all entailment based algorithms for explicit folding/unfolding, splitting, and lemma application. Alternatively, one could try to embed our logic in an existing theorem prover and define tactics (or tacticals) corresponding to our proof search algorithms. This would automatically provide flexibility regarding extensions with new advanced hints.

In terms of reducing the burden on the human verifier, it may be desirable to integrate some form of invariant and specification synthesis to alleviate the need to provide loop invariants and pre- and post-conditions. It would be interesting to explore whether techniques such as "shape analysis", which have already been demonstrated to lend themselves to separation logic [20,11], still apply in a higher-order store environment. In particular, whether one can automatically infer the

"deepframe" invariants that currently must be provided to call and code-store commands.

Acknowledgements

We would like to thank the anonymous referees for their helpful suggestions and comments. This research has been sponsored by the *Engineering and Physical Sciences Research Council*, grant EP/G003173/1: *"From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs"*.

References

1. The Crowfoot website, 2011. `www.sussex.ac.uk/informatics/crowfoot`.
2. Olav Beckmann, Alastair Houghton, Michael R. Mellor, and Paul H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.
3. Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*, pages 301–312, 2009.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
5. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
6. B. Biering, L. Birkedal, and N. Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
7. Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed Kripke models over recursive worlds. In *POPL'11*, pages 119–132. IEEE, Jan 2011.
8. Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5), 2006.
9. Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands, November 2013.
10. Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *PLDI*, pages 66–77, 2007.
11. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *ACM SIGPLAN Notices*, 44(1):289–300, 2009.
12. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *ICFP*, pages 418–430. ACM, 2011.
13. Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Formal reasoning about runtime code update. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE Workshops*, pages 134–138. IEEE, 2011.
14. Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Crowfoot: A verifier for higher-order store programs. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 2012.
15. Nathaniel Charlton and Bernhard Reus. A deeper understanding of the deep frame axiom. Extended abstract, presented at LOLA (Syntax and Semantics of Low Level Languages), 2010.
16. Nathaniel Charlton and Bernhard Reus. Specification patterns and proofs for recursion through the store. In *FCT*, pages 310–321, 2011.
17. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
18. Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 234–245. ACM, 2011.

19. Adam Chlipala, J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 79–90. ACM, 2009.
20. Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
21. Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.
22. Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM*, pages 386–401, 2011.
23. Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.
24. Bryan Henderson. Linux loadable kernel module HOWTO (v1.09), 2006. Available online. `http://tldp.org/HOWTO/Module-HOWTO/`.
25. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer Berlin / Heidelberg, 1971.
26. Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. In *LICS*, pages 270–279, 2005.
27. Ben Horsfall. Automated reasoning for reflective programs, 2014. PhD thesis.
28. Ben Horsfall, Nathaniel Charlton, and Bernhard Reus. Verifying the reflective visitor pattern. In *FtFJP*, pages 27–34, 2012.
29. Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011.
30. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, pages 304–311, 2010.
31. Wonyeol Lee and Sungwoo Park. A proof system for separation logic with magic wand. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 477–490, New York, NY, USA, 2014. ACM.
32. Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.
33. Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *POPL*, pages 320–333, 2006.
34. François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, pages 331–340, Pittsburgh, Pennsylvania, June 2008.
35. David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.
36. Bernhard Reus and Jan Schwinghammer. Separation logic for higher-order store. In *CSL*, pages 575–590, 2006.
37. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
38. Jan J. M. M. Rutten. Elements of generalized ultrametric domain theory. *Theoretical Computer Science*, 170(1–2):349–381, December 1996.
39. M. Schwerhoff and A. J. Summers. Lightweight support for magic wands in an automatic verifier. Technical report, ETH Zurich, 2014.
40. Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, pages 440–454, 2009.
41. Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rule for higher-order store. *Logical Methods in Computer Science*, 7(3), September 2011.
42. Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. A semantic foundation for hidden state. In *FOSSACS*, pages 2–17, 2010.
43. Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

Appendices

In the Appendices we present a collection of all the rules used (and not always discussed in the main text) as well as selected soundness proofs of Crowfoot's proof search rules in terms of the "core" Hoare rules of the logic for higher-order store.

A Core rules

A.1 Rules for Hoare triples

A.1.1 Syntax Driven Rules for Triples

For the sake of brevity, we often drop the contexts Π and/or Σ, resp., where they do not play any role.

$$\textsc{assign}\quad \{P\}\, x := e\, \{\exists x' \,.\, x = e[x\backslash x'] \star P[x\backslash x']\}$$

$$\textsc{lookup}\quad \{P \star e_A \mapsto e'\}\, x := [e_A]\, \{\exists x' \,.\, x = e'[x\backslash x'] \star (e_A \mapsto e')[x\backslash x'] \star P[x\backslash x']\}$$

$$\textsc{heap-assign}\quad \{e \mapsto \_\}\, [e_A] := e'\, \{e_A \mapsto e'\}$$

$$\textsc{heap2heap-assign}\quad \{e \mapsto \mathscr{C} \star e' \mapsto \_\}\, [e_A] := [e'_A]\, \{e_A \mapsto \mathscr{C} \star e'_A \mapsto \mathscr{C}\}$$

$$\textsc{new}\quad \{P\}\, x := \mathsf{new}\ e\, \{\exists x' \,.\, x \mapsto e[x\backslash x'] \star P[x\backslash x']\} \qquad \textsc{dispose}\quad \{e_A \mapsto \_\}\, \mathsf{dispose}\ e_A\, \{\mathsf{emp}\}$$

$$\textsc{Call}\quad \frac{\Pi \Vdash (\forall \mathbf{y}.\, \{A\}\, k(\mathbf{p})\, \{B\}) \Rightarrow \{P\}\, k(\mathbf{e_V})\, \{Q\}}{\Pi; \Sigma, \{A\}\, \mathcal{F}(\mathbf{p})\, \{B\} \Vdash \{P\}\, \mathsf{call}\ \mathcal{F}(\mathbf{e_V})\, \{Q\}}\ \mathbf{y} = fv(A, \mathbf{p}, B) \qquad k \text{ fresh}$$

$$\textsc{Eval}\quad \frac{P \Rightarrow e_A \mapsto \{P\} \cdot (\mathbf{e_V})\, \{Q\} \star \mathit{true}}{\{P\}\, \mathsf{eval}\ [e_A](\mathbf{e_V})\, \{Q\}} \qquad \textsc{skip}\quad \{P\}\, \mathsf{skip}\, \{P\}$$

$$\textsc{StoreProc}\quad \{A\}\, \mathcal{F}(\mathbf{p})\, \{B\} \Vdash \{e_A \mapsto \_\}\, [e_A] := \mathcal{F}(\mathbf{t})\, \{e_A \mapsto (\forall \mathbf{p}|_U, \mathbf{y}.\ \{A\} \cdot (\mathbf{p}|_U)\, \{B\})\, [\mathbf{p}|_{I\setminus U} \backslash \mathbf{t}|_{I\setminus U}]\}$$

where $|\mathbf{t}| = |\mathbf{p}|$ and t_i either value expression a_i or $\_$; $\mathbf{y} = fv(A, B) - \mathbf{p}$;

$$\mathbf{p} = (p_i)_{i \in I}; \quad U = \{i \in I \mid t_i = \_\} \quad \mathbf{p}|_X = (p_i)_{i \in I \cap X}$$

$$\textsc{scomp}\quad \frac{\Pi; \Gamma \Vdash \{P\}\, C_1\, \{R\} \quad \Pi; \Gamma \Vdash \{R\}\, C_2\, \{Q\}}{\Pi; \Gamma \Vdash \{P\}\, C_1; C_2\, \{Q\}}$$

$$\textsc{if}\quad \frac{\Pi; \Gamma \Vdash \{P \wedge e_V = e'_V\}\, C_1\, \{Q\} \quad \Pi; \Gamma \Vdash \{P \wedge e_V \neq e'_V\}\, C_2\, \{Q\}}{\Pi; \Gamma \Vdash \{P\}\, \mathsf{if}\ e_V = e'_V\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\, \{Q\}}$$

$$\textsc{while1}\quad \frac{\Pi; \Gamma \Vdash \{I \wedge e_V = e'_V\}\, C\, \{I\}}{\Pi; \Gamma \Vdash \{I\}\, \mathsf{while}\ e_V = e'_V\ \mathsf{do}\ C\, \{I \wedge e_V \neq e'_V\}} \qquad \textsc{while2}\quad \frac{\Pi; \Gamma \Vdash \{I \wedge e_V \neq e'_V\}\, C\, \{I\}}{\Pi; \Gamma \Vdash \{I\}\, \mathsf{while}\ e_V \neq e'_V\ \mathsf{do}\ C\, \{I \wedge e_V = e'_V\}}$$

A.1.2 Non-syntax driven Rules for Triples

$$\textsc{Disjunction-Precond}\quad \frac{\Pi;\Gamma \Vdash \{P_1\}\,C\,\{Q\} \qquad \Pi;\Gamma \Vdash \{P_2\}\,C\,\{Q\}}{\Pi;\Gamma \Vdash \{P_1 \vee P_2\}\,C\,\{Q\}}$$

$$\textsc{Skolemize}\quad \Pi;\Gamma \Vdash \{P[v\backslash v_0]\}\,C\,\{Q\} \Rightarrow \{\exists v.P\}\,C\,\{Q\}\ v_0 \text{ is fresh} \qquad \textsc{False}\quad \{\mathit{false}\}\,C\,\{Q\}$$

$$\textsc{ShallowFrame}\quad \frac{\Pi;\Gamma \Vdash \{P\}\,C\,\{Q\}}{\Pi;\Gamma \Vdash \{P \star R\}\,C\,\{Q \star R\}}\ \mathit{mod}(C) \cap \mathit{fv}(R) = \emptyset$$

A.1.3 Rules that use Predicate or Procedure Declaration Context

$$\textsc{Consequence}\quad \frac{\Pi;\Gamma \Vdash \{P'\}\,C\,\{Q'\}}{\Pi;\Gamma \Vdash \{P\}\,C\,\{Q\}}\ \Pi \Vdash P \Rightarrow P' \text{ and } \Pi \Vdash Q' \Rightarrow Q$$

$$\textsc{deepframe}\quad \frac{\Pi;\Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\}\,, \{P \circ \Phi\}\,\mathcal{F}(\mathbf{p})\,\{Q \circ \Phi\} \ \Vdash\ B}{\Pi;\Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \ \Vdash\ B}$$

A.2 Rules for Entailments between Assertions

For the sake of brevity, instead of $\Pi \vdash A \Rightarrow B$ we simply write $A \Rightarrow B$ where Π does not play any role (and analogously for $\Leftrightarrow$ and $\Leftarrow$).

A.2.1 Separation Logic Axioms

$$\star\textsc{-Commutative}\quad A * \Phi \Leftrightarrow \Phi * A \qquad \star\textsc{-Associative}\quad A * (B * \Phi) \Leftrightarrow (A * B) * \Phi \qquad \mathsf{emp}\textsc{-Unit}\quad A * \mathsf{emp} \Leftrightarrow A$$

$$\star\textsc{-Monotonicity}\quad \frac{\Pi \vdash \Phi \Rightarrow \Theta}{\Pi \vdash \Phi \star A \Rightarrow \Theta \star A} \qquad \exists\textsc{-}*\textsc{Distribution}\quad (\exists v.\Phi) \star \Theta \ \Leftrightarrow\ \exists v.(\Phi \star \Theta) \quad \text{if } v \text{ is not free in } \Theta$$

$$\mapsto\textsc{-Group}\quad \Pi \vdash e_A \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_n \Leftrightarrow e_A \mapsto \mathscr{C}_1 \ \star\ e_A + 1 \mapsto \mathscr{C}_1 \ \star\ \ldots\ \star\ e_A + n - 1 \mapsto \mathscr{C}_n$$

$$\textsc{Purify}\quad \Phi \ \Leftrightarrow\ \mathit{purify}(\Phi) \star \Phi \qquad \textsc{Closure}\quad \Phi \ \Leftrightarrow\ \mathit{closure}(\Phi)$$

$$\star\textsc{-SplitPureLeft}\quad A \star \Phi \Rightarrow A \quad \text{if } \Phi \text{ is pure} \qquad \star\textsc{-SplitPureRight}\quad A \wedge (\mathit{true} \star \Phi) \Rightarrow A \star \Phi \quad \text{if } \Phi \text{ is pure}$$

A.2.2 Distribution laws for deep framing

$$\otimes\textsc{-Triple}\quad \{P\}\,e(\mathbf{e})\,\{Q\} \otimes R \ \Leftrightarrow\ \{P \circ R\}\,e(\mathbf{e})\,\{Q \circ R\}$$

$$\otimes\textsc{-Connectives}\quad (P \oplus Q) \otimes R \Leftrightarrow (P \otimes R) \oplus (Q \otimes R) \qquad \text{where } \oplus \in \{\vee, \star\}$$

$$\otimes\textsc{-Quantors}\quad (\kappa v.P) \otimes R \ \Leftrightarrow\ \kappa v.(P \otimes R) \qquad \text{where } \kappa \in \{\forall, \exists\}, \text{ provided } v \text{ not free in } R$$

$$\circ\textsc{-Definition}\quad (P \circ R) \ \Leftrightarrow\ (P \otimes R) \star R$$

The following rule is used to prove the Plotkin Lemma needed to show Lemma 7 and Lemma 8.

$$\textsc{RUnique}\quad \frac{\forall \mathbf{y}.\,(R[\mathrm{X}\backslash\mathrm{P}])(\mathbf{y}) \Leftrightarrow \mathrm{P}(\mathbf{y}) \qquad \forall \mathbf{y}.\,(R[\mathrm{X}\backslash\mathrm{Q}])(\mathbf{y}) \Leftrightarrow \mathrm{Q}(\mathbf{y})}{\mathrm{P}(\mathbf{y}) \Leftrightarrow \mathrm{Q}(\mathbf{yz})}$$

where R is an assertion with a free predicate variable X (used with the appropriate arity) from the pattern as described in [16] (so in particular it cannot be X itself) that guarantees that R has a fixpoint.

A.2.3 Rules that use Predicate declaration Context

$$\textsc{PredLeft}\quad \forall \mathbf{y}.\, \Pi, (\mathrm{X}(\mathbf{y}) \Leftrightarrow P) \Vdash P \Rightarrow \mathrm{X}(\mathbf{y}) \qquad \textsc{PredRight}\quad \Pi, (\mathrm{X}(\mathbf{y}) \Leftrightarrow P) \Vdash \mathrm{X}(\mathbf{y}) \Rightarrow P$$

A.2.4 Inductive Rules for Inductive Predicates

$$\textsc{ListInduction}\quad \frac{\begin{array}{c}\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := Q \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \\ \Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow Q) \Vdash s = t \star \alpha = \emptyset \Rightarrow P \\ \Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow Q) \Vdash \left(\begin{array}{l} P[s, \alpha\backslash n, \beta] \ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \star \\ A_1 \star \cdots \star A_N \ \star\ \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \end{array}\right) \Rightarrow P \end{array}}{\Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow Q) \Vdash \mathrm{L}(s,t,\alpha) \Rightarrow P} \quad n, \mathbf{v}, \beta \notin fv(P)$$

$$\textsc{Join}\quad \frac{\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := P \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k])}{\Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow P) \Vdash \mathrm{L}(e_1, e, e_\alpha) \star \mathrm{L}(e, e_2, e_\beta) \Rightarrow \mathrm{L}(e_1, e_2, e_\alpha \cup e_\beta)}$$

$$\textsc{Split}\quad \frac{\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := P \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_M], [A_1, \ldots, A_N], [E_1, \ldots, E_k])}{\begin{array}{l}\Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow P) \Vdash (e_1, \ldots, e_k) \in \hat{e}_\gamma \ \star\ \mathrm{L}(\hat{e}_1, \hat{e}_2, \hat{e}_\gamma) \Rightarrow \\ \qquad \exists s, n, \mathbf{v}, \alpha, \beta. \left(\begin{array}{l} \mathrm{L}(\hat{e}_1, s, \alpha) \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, \hat{e}_2, \beta) \\ \star\ \hat{e}_\gamma = \alpha \cup \{(e_1, \ldots, e_k)\} \cup \beta \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{array}\right)\end{array}}$$

Let us prove soundness of the Split rule. We will prove the particular case where E_1 is the variable s used as the address of the first list node in the segment, which ensures that L is splittable. Other cases are similar.

Proof We use the (ListInduction) rule, taking P to be

$$(e_1, \ldots, e_k) \in \alpha \Rightarrow \left(\begin{array}{l} \exists a, n, \mathbf{v}, \alpha_1, \alpha_2. \\ \quad \mathrm{L}(s, a, \alpha_1) \\ \star\ a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, z, \alpha_2) \\ \star\ \alpha = \alpha_1 \cup \{(e_1, \ldots, e_k)\} \cup \alpha_2 \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{array}\right) \tag{20}$$

Thus for the "base case" we need to prove

$$\Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow Q) \Vdash s = t \star \alpha = \emptyset \Rightarrow \left(\begin{array}{l} (e_1, \ldots, e_k) \in \alpha \Rightarrow \end{array}\ \begin{array}{l} \exists a, n, \mathbf{v}, \alpha_1, \alpha_2. \\ \quad \mathrm{L}(s, a, \alpha_1) \\ \star\ a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, z, \alpha_2) \\ \star\ \alpha = \alpha_1 \cup \{(e_1, \ldots, e_k)\} \cup \alpha_2 \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{array}\right)$$

which is easily seen to hold because $\alpha = \emptyset$ together with $(e_1, \ldots, e_k) \in \alpha$ is inconsistent. Then for the inductive case we need to show

$$\Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow Q) \Vdash$$

$$\begin{array}{l} P[s, \alpha\backslash n, \beta] \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \end{array} \quad \Rightarrow \quad \left(\begin{array}{ll} & \exists a, n, \mathbf{v}, \alpha_1, \alpha_2. \\ & \quad \mathrm{L}(s, a, \alpha_1) \\ & \star\ a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ (e_1, \ldots, e_k) \in \alpha \Rightarrow & \star\ A_1 \star \cdots \star A_N \\ & \star\ \mathrm{L}(n, z, \alpha_2) \\ & \star\ \alpha = \alpha_1 \cup \{(e_1, \ldots, e_k)\} \cup \alpha_2 \\ & \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{array} \right)$$

which amounts to

$$\Pi, (\mathrm{L}(s,t,\alpha) \Leftrightarrow Q) \Vdash$$

$$\begin{array}{l} \left(\begin{array}{ll} s = t \star \alpha = \emptyset \quad \vee & \left(\begin{array}{l} \exists n, \mathbf{v}, \beta\ . \\ \quad s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \quad \star\ A_1 \star \cdots \star A_N \\ \quad \star\ \mathrm{P}(n, t, \beta) \\ \quad \star\ \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \end{array} \right) [s, \alpha\backslash n, \beta] \end{array} \right) \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \alpha = \{(E_1, \ldots, E_k)\} \cup \beta \\ \star\ (e_1, \ldots, e_k) \in \alpha \end{array} \quad \Rightarrow \quad \begin{array}{l} \exists a, n, \mathbf{v}, \alpha_1, \alpha_2. \\ \quad \mathrm{L}(s, a, \alpha_1) \\ \star\ a \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_M, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, z, \alpha_2) \\ \star\ \alpha = \alpha_1 \cup \{(e_1, \ldots, e_k)\} \cup \alpha_2 \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{array}$$

A.2.5 Rules for entailment between behavioural specifications

Where these rules do *not* require Π it will be omitted.

ShallowFrameProcedures
$$\{P\}\, k(\mathbf{e_V})\, \{Q\} \Rightarrow \{P \star R\}\, k(\mathbf{e_V})\, \{Q \star R\}$$

ConsequenceProcedures
$$\frac{\Pi \Vdash P \Rightarrow P' \qquad \Pi \Vdash Q' \Rightarrow Q}{\Pi \Vdash \{P'\}\, k(\mathbf{e_V})\, \{Q'\} \Rightarrow \{P\}\, k(\mathbf{e_V})\, \{Q\}}$$

Disjunction
$$\frac{\Pi \Vdash A \Rightarrow \{P_1\}\, k(\mathbf{e_V})\, \{Q\} \qquad \Pi \Vdash A \Rightarrow \{P_2\}\, k(\mathbf{e_V})\, \{Q\}}{\Pi \Vdash A \Rightarrow \{P_1 \vee P_2\}\, k(\mathbf{e_V})\, \{Q\}}$$

Skolem
$$\{\Phi[v\backslash v_0]\}\, k(\mathbf{e_V})\, \{Q\} \Rightarrow \{\exists v.\Phi\}\, k(\mathbf{e_V})\, \{Q\} \quad v_0 \text{ fresh}$$

B Symbolic execution rules

B.1 Atomic statements

$$\textsc{Assign}\quad \frac{\Pi;\Gamma \triangleright \{x = e_v[x\backslash x'] \star \Upsilon[x\backslash x']\}\, C\, \{Q\}}{\Pi;\Gamma \triangleright \{\Upsilon\}\, x := e_v;\ C\, \{Q\}}\ x' \text{ fresh}$$

$$\textsc{Lookup}\quad \frac{\begin{array}{c}\mathit{purify}(\Upsilon) \vdash_{SMT} E_A = (e'_A + o)\\ \Pi;\Gamma \triangleright \{x = (e_V[x\backslash x'])\ \star \Upsilon \star e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e_V, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\}\end{array}}{\Pi;\Gamma \triangleright \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e_V, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, x := [E_A];\ C\, \{Q\}}\ x' \text{ fresh}$$

$$\textsc{Mutate}\quad \frac{\begin{array}{c}\mathit{purify}(\Upsilon) \vdash_{SMT} E_A = (e'_A + o)\\ \Pi;\Gamma \triangleright \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, x, \mathscr{C}_{o+1} \ldots, \mathscr{C}_n\}\, C\, \{Q\}\end{array}}{\Pi;\Gamma \triangleright \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\}\, [E_A] := x;\ C\, \{Q\}}$$

$$\textsc{Copy}\quad \frac{\begin{array}{c}\mathit{purify}(\Upsilon) \vdash_{SMT} E_1 = (e_A + o) \qquad \mathit{purify}(\Upsilon) \vdash_{SMT} E_2 = (e'_A + o')\\ \Pi;\Gamma \triangleright \left\{\begin{array}{l} e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, \mathscr{C}'_{o'}, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_m \star \\ e'_A \mapsto \mathscr{C}'_0, \ldots, \mathscr{C}'_n \star \Upsilon \end{array}\right\}\ C\ \{Q\}\end{array}}{\Pi;\Gamma \triangleright \left\{\begin{array}{l} e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_m \star \\ e'_A \mapsto \mathscr{C}'_0, \ldots, \mathscr{C}'_n \star \Upsilon \end{array}\right\}\ [E_1] := [E_2];\ C\ \{Q\}}\ 0 \le o \le m, 0 \le o' \le n$$

$$\textsc{New}\quad \frac{\Pi;\Gamma \triangleright \{\Upsilon[x\backslash x'] \star x \mapsto (e_0, \ldots, e_n)[x\backslash x']\}\, C\, \{Q\}}{\Pi;\Gamma \triangleright \{\Upsilon\}\, x := \mathsf{new}\ e_0, \ldots, e_n;\ C\, \{Q\}}\ x' \text{ fresh}$$

$$\textsc{Dispose}\quad \frac{\mathit{purify}(\Upsilon) \vdash_{SMT} e_A = (e'_A + o) \qquad \Pi;\Gamma \triangleright \left\{\begin{array}{l} \Upsilon \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1} \star \\ (e'_A + o + 1) \mapsto \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n \end{array}\right\}\ C\ \{Q\}}{\Pi;\Gamma \triangleright \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\}\, \mathsf{dispose}\ e_A;\ C\, \{Q\}}$$

$$0 \le o \le n$$

$$\textsc{Call}\quad \frac{\begin{array}{c}(\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\})\, [\mathbf{b}\backslash\mathbf{y}]\ \ \otimes \Psi \vdash_{\text{find-post}} \{\Upsilon\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi \right\}\\ \Pi; \forall \mathbf{a}, \mathbf{b}.\, \{P\}\, \mathcal{F}(\mathbf{t})\, \{Q\}\,, \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi[\mathbf{v_i}\backslash\mathbf{v'_i}] \right\} C\, \{Q'\}\end{array}}{\Pi; \forall \mathbf{a}, \mathbf{b}.\, \{P\}\, \mathcal{F}(\mathbf{t})\, \{Q\}\,, \Gamma \triangleright \{\Upsilon\}\, \mathsf{call}\ \mathcal{F}(\mathbf{t'})\ \text{`}\mathbf{b} = \mathbf{y}\text{'}\ \mathsf{deepframe}\ \Psi;\ C\, \{Q'\}}\ \mathbf{v'_i} \text{ fresh}$$

EvalUnguided

$$\frac{\begin{array}{c}\Pi : \Upsilon \vdash_{\text{find-tr}} \exists \mathbf{u}.\, e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure} \\ \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\}\, [\mathbf{u}\backslash\mathbf{w}] \vdash_{\text{find-post}} \{\Upsilon \star R^{pure}[\mathbf{u}\backslash\mathbf{w}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Phi_i \right\} \\ \Pi; \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i}\backslash\mathbf{v'_i}] \right\} C\, \{Q'\}\end{array}}{\Pi; \Gamma \triangleright \{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'});\ C\, \{Q'\}}\ \mathbf{v'_i}, \mathbf{w} \text{ fresh}$$

Eval

$$\frac{\begin{array}{c}\Pi : \Upsilon \vdash^{\mathbf{G}}_{\text{find-tr}} \exists \mathbf{y}.\, e_A \mapsto \forall \mathbf{a}, \mathbf{b}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure} \\ (\forall \mathbf{p}.\, \{A\} \cdot (\mathbf{s})\, \{B\})\, [\mathbf{q}\backslash\mathbf{E}] \vdash_{\text{find-post}} \left\{\Upsilon \star R^{pure}[\mathbf{y}\backslash\mathbf{y'}]\right\} \cdot (\mathbf{s'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{w}_i.\Theta_i \right\} \\ \mathit{purify}(\mathit{closure}(\Theta_i[\mathbf{w}_i\backslash\mathbf{w'_i}])) \vdash_{SMT} \mathit{false} \text{ for all } i \neq k \\ (\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\})\, [\mathbf{b}\backslash\mathbf{e}][\mathbf{y}\backslash\mathbf{y'}] \vdash_{\text{find-post}} \left\{\Theta_k[\mathbf{w}_k\backslash\mathbf{w'_k}]\right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Phi_i \right\} \\ \Pi; \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i}\backslash\mathbf{v'_i}] \right\} C\, \{Q'\}\end{array}}{\Pi; \forall \mathbf{p}, \mathbf{q}.\, \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \ \triangleright \{\, \Upsilon \,\} \quad \begin{array}{l}\mathsf{eval}\ [e_A](\mathbf{t'})\ \text{`}\mathbf{b} = \mathbf{e} \\ \quad \mathsf{before\ lookup}\ \mathbf{G} \\ \quad \mathsf{after\ lookup}\ \mathcal{L}(\mathbf{s'})\ \mathbf{q} = \mathbf{E}\ \text{'};\ C\end{array} \quad \{\, Q' \,\}}$$

$\mathbf{y'}, \mathbf{v'_i}, \mathbf{w'_i}$ fresh
$\mathcal{L}$ is a lemma procedure

Skip1

$$\frac{\Pi; \Gamma \triangleright \{\Upsilon\}\, C\, \{Q\}}{\Pi; \Gamma \triangleright \{\Upsilon\}\, \mathsf{skip};\ C\, \{Q\}}$$

StoreCode

$$\frac{\begin{array}{c}S = (\forall \mathbf{p}|_U, \mathbf{y}.\ \{P \circ \Psi\} \cdot (\mathbf{p}|_U)\, \{Q \circ \Psi\})\, [\mathbf{p}|_{I\backslash U}\backslash\mathbf{t}|_{I\backslash U}] \\ \mathit{purify}(\Upsilon) \vdash_{SMT} e_A = e'_A + o \\ \Pi; \Gamma, \{P\}\, \mathcal{F}(\mathbf{p})\, \{Q\} \triangleright \left\{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\right\} C\, \{Q'\}\end{array}}{\Pi; \Gamma, \{P\}\, \mathcal{F}(\mathbf{p})\, \{Q\} \triangleright \{\, \Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n \,\} \quad \begin{array}{l}[e_A] := \mathcal{F}(\mathbf{t}) \\ \text{`}\mathsf{deepframe}\ \Psi\text{'};\ C\end{array} \quad \{\, Q' \,\}}$$

where $|\mathbf{t}| = |\mathbf{p}|$ and t_i either value expression a_i or _; $\mathbf{y} = \mathit{fv}(P \circ \Psi, Q \circ \Psi) - \mathbf{p}$;

$$\mathbf{p} = (p_i)_{i \in I}; \quad U = \{i \in I \mid t_i = \_\} \quad \mathbf{p}|_X = (p_i)_{i \in I \cap X} \text{ for any } X$$

StoreCodeGuided

$$\frac{\begin{array}{c}\Pi : ((\Phi \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \ \vdash^{\emptyset}_{G^{\text{pre}}}\ X(\mathbf{E}) \star \Phi' \\ \Pi : (((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}]) \star \mathsf{emp}\ \dashv\vdash^{\emptyset}_{G^{\text{post}}}\ Y(\mathbf{e}) \star \Theta' \\ S = \left(\begin{array}{c} \{\, \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \,\} \\ \forall \mathbf{p}|_U, \mathbf{y}. \qquad \cdot(\mathbf{p}|_U) \\ \{\, \exists \mathbf{w}, \mathbf{a'}.\ (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}] \,\} \end{array} \right) [\mathbf{p}|_{I\backslash U}\backslash\mathbf{t}|_{I\backslash U}] \\ \mathit{purify}(\Upsilon) \vdash_{SMT} e_A = e'_A + o \\ \Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\, \mathcal{F}(\mathbf{p})\, \{\exists \mathbf{w}.\Theta\} \triangleright \left\{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\right\} C\, \{Q'\}\end{array}}{\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\, \mathcal{F}(\mathbf{p})\, \{\exists \mathbf{w}.\Theta\} \triangleright \{\, \Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n \,\} \quad \begin{array}{l}[e_A] := \mathcal{F}(\mathbf{t}) \\ \text{`}\mathsf{deepframe}\ \exists \mathbf{a}.\ \Upsilon' \\ \mathsf{pre}\ G^{\text{pre}} \\ \mathsf{post}\ G^{\text{post}}\ \text{'};\ C\end{array} \quad \{\, Q' \,\}}$$

where $|\mathbf{t}| = |\mathbf{p}|$ and t_i either value expression a_i or _;
$\mathbf{y} = \mathit{fv}(S\text{-precondition}, S\text{-postcondition}) - \mathbf{p}$; $\mathbf{v'}, \mathbf{w'}, \mathbf{a'}, \mathbf{v''}, \mathbf{w''}$ fresh
$\mathbf{p} = (p_i)_{i \in I}; \quad U = \{i \in I \mid t_i = \_\} \quad \mathbf{p}|_X = (p_i)_{i \in I \cap X}$ for any X

B.2 Ghost statements

$$\textsc{GhostFold}\quad \frac{\Pi : \Upsilon \;\vdash^{I}_{G}\; X(\mathbf{e}) \star \Theta \qquad \Pi; \Gamma \triangleright \{X(\mathbf{e}) \star \Theta\}\, C\, \{Q\}}{\Pi; \Gamma \triangleright \{\Upsilon\}\, \mathsf{ghost\ fold}\ X(\mathbf{p})\ \mathbf{b} = \mathbf{y};\ C\, \{Q\}}\ G = \mathsf{fold}\ X(\mathbf{p})\ \mathbf{b} = \mathbf{y}$$

$$\textsc{GhostUnfold}\quad \frac{\Pi, \mathrm{X}(\mathbf{x}) \Leftrightarrow (\exists \mathbf{v}_1.\Phi_1) \vee \ldots \vee (\exists \mathbf{v}_n.\Phi_n); \Gamma \triangleright \left\{ \bigvee_{i=1}^{n} \Upsilon \star \Phi_i[\mathbf{v}_i \backslash \mathbf{w}_i][\mathbf{x} \backslash \mathbf{e}] \right\} C\, \{Q\}}{\Pi, \mathrm{X}(\mathbf{x}) \Leftrightarrow (\exists \mathbf{v}_1.\Phi_1) \vee \ldots \vee (\exists \mathbf{v}_n.\Phi_n); \Gamma \triangleright \{\Upsilon \star X(\mathbf{e})\}\, \mathsf{ghost\ unfold}\ X(\mathbf{e'});\ C\, \{Q\}}$$

$$\mathbf{w}_1, \ldots, \mathbf{w}_n \text{ fresh} \qquad e'_i \in \{e_i, ?\}$$

$$\textsc{GhostJoin}\quad \frac{\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := P \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_K], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \quad \mathit{purify}(\Upsilon) \vdash_{SMT} e = e' \qquad \Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow P \triangleright \{\Upsilon \star \mathrm{L}(e_1, e_2, e_\alpha \cup e_\beta)\}\, C\, \{Q\}}{\Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow P \triangleright \{\Upsilon \star \mathrm{L}(e_1, e, e_\alpha) \star \mathrm{L}(e', e_2, e_\beta)\}\, \mathsf{ghost\ join}\ \mathrm{L}\ e_1;\ C\, \{Q\}}$$

$$\textsc{GhostSplit}\quad \frac{\begin{array}{c}\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := P \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_K], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \\ \Upsilon \vdash^{I} \exists c_1, \ldots, c_k\ .\ (q_1, \ldots, q_k) \in \hat{e}_\gamma \star \Theta \\ \Pi, \mathrm{L}(\mathbf{x}) \Leftrightarrow \Psi; \Gamma \triangleright \left\{ \Upsilon \star \begin{pmatrix} \mathrm{L}(\hat{e}, s, \alpha) \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_K, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, \hat{e}', \beta) \\ \star\ \hat{e}_\gamma = \alpha \cup \{(e_1, \ldots, e_k)\} \cup \beta \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{pmatrix} [s, n, \mathbf{v} \backslash \mathbf{w}] \right\}\ C\ \ \{\, Q \,\}\end{array}}{\Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow P; \Gamma \triangleright \{\Upsilon \star \mathrm{L}(\hat{e}, \hat{e}', \hat{e}_\gamma)\}\, \mathsf{ghost\ split}\ \mathrm{L}\ \hat{e}\ (p_1, \ldots, p_k);\ C\, \{Q\}}$$

$$\begin{array}{l}\alpha, \beta, \mathbf{w} \text{ fresh},\ c_1, \ldots, c_k \text{ fresh}, \\ \text{each } p_i \text{ is a value expression or ?}, \\ \text{each } q_i \text{ is } \begin{cases} c_i & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}, \\ \text{each } e_i \text{ is } \begin{cases} I(c_i) & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}\end{array}$$

B.3 Extraordinary rules

$$\textsc{Disj}\quad \frac{\Pi; \Gamma \triangleright \{\Psi_1\}\, C\, \{Q\} \qquad \cdots \qquad \Pi; \Gamma \triangleright \{\Psi_n\}\, C\, \{Q\}}{\Pi; \Gamma \triangleright \{\Psi_1 \vee \cdots \vee \Psi_n\}\, C\, \{Q\}}$$

$$\textsc{Skip2}\quad \frac{\Upsilon \;\vdash^{I}\; \exists \mathbf{w}.\Phi_i[\mathbf{v}_i \backslash \mathbf{w}] \star \Theta^{pure}}{\{\Upsilon\}\, \mathsf{skip}\, \{\exists \mathbf{v}_1.\Phi_1 \vee \ldots \vee \exists \mathbf{v}_n.\Phi_n\}}\ \mathbf{w} \text{ fresh}$$

$$\textsc{Incons}\quad \frac{\mathit{purify}(\mathit{closure}(\Upsilon)) \vdash_{SMT} \mathit{false}}{\{\Upsilon\}\, C\, \{Q\}}$$

$$\textsc{AddSkip}\quad \frac{\Pi; \Gamma \triangleright \{\Upsilon\}\, C\ ;\ \mathsf{skip}\, \{Q\}}{\Pi; \Gamma \triangleright \{\Upsilon\}\, C\, \{Q\}}\ C \text{ is not a sequential composition}$$

C Soundness of symbolic execution rules

To prove soundness of the low-level deterministic symbolic execution rules in the following we will use the already shown sound high-level rules.

C.1 Soundness of Lookup

We derive the low-level lookup rule from the high-level one in Section 4. For the sake of a simpler presentation we omit the context $\Pi;\Gamma$. The low-level rule is the following:

$$\textsc{Lookup}\quad \frac{\mathit{purify}(P) \vdash_{SMT} E = (G+o) \qquad \{x = (e[x\backslash x']) \star (P \star G \mapsto \mathscr{C}_0,\ldots,\mathscr{C}_{o-1},e,\mathscr{C}_{o+1},\ldots,\mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\}}{\{P \star G \mapsto \mathscr{C}_0,\ldots,\mathscr{C}_{o-1},e,\mathscr{C}_{o+1},\ldots,\mathscr{C}_n\}\, x := [E];\ C\, \{Q\}}$$

1. x' fresh

Proof. We can assume that:

(a) x' is fresh
(b) $\mathit{purify}(P) \vdash_{SMT} E = (G+o)$
(c) $\{x = (e[x\backslash x']) \star (P \star G \mapsto \mathscr{C}_0,\ldots,\mathscr{C}_{o-1},e,\mathscr{C}_{o+1},\ldots,\mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\}$

We are then required to show that that we can derive the conclusion of the low-level rule from the high-level axiom. With some renaming of variables (e, e', E), and instantiating P, the axiom becomes:

$$\left\{\begin{array}{l} P \star G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star E \mapsto e \end{array}\right\}\ x := [E]\ \left\{\begin{array}{l} \exists x'.\ x = e[x\backslash x'] \star (E \mapsto e)[x\backslash x'] \star \\ \left(\begin{array}{l} P \star G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \end{array}\right)[x\backslash x'] \end{array}\right\}$$

We use the following lemma:

Lemma 9 *If* $\mathit{purify}(P) \Rightarrow R^{pure}$ *then* $P \Leftrightarrow P \star R^{pure}$.

Proof This can easily be derived using (Purify), ($\star$-SplitPureLeft) and the axioms of Separation Logic.

Using (b), and Theorem 3 (soundness of SMT), we get:

$$\mathit{purify}(P) \Rightarrow E = (G+o) \tag{21}$$

providing the lhs and rhs are both pure. The left is pure by definition of $\mathit{purify}(\_)$, and the right is pure by definition of $=$.

Using (a), (c), (Skolem), and some renaming we get:

$$\begin{array}{l} \{x = (e[x\backslash x']) \star (P \star G \mapsto \mathscr{C}_0,\ldots,\mathscr{C}_{o-1},e,\mathscr{C}_{o+1},\ldots,\mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\} \\ \text{rename } x' \text{ with fresh } x_0 \\ \{(x = (e[x\backslash x']) \star (P \star G \mapsto \mathscr{C}_0,\ldots,\mathscr{C}_{o-1},e,\mathscr{C}_{o+1},\ldots,\mathscr{C}_n)[x\backslash x'])[x'\backslash x_0]\}\, C\, \{Q\} \\ \textsc{Skolem} \\ \{\exists x'.\ x = (e[x\backslash x']) \star (P \star G \mapsto \mathscr{C}_0,\ldots,\mathscr{C}_{o-1},e,\mathscr{C}_{o+1},\ldots,\mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\} \end{array} \tag{22}$$

We then reason as follows:

$$\left\{\begin{array}{l} P \star G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star E \mapsto e \end{array}\right\} \; x := [E] \; \left\{\begin{array}{l} \exists x'.\, x = e[x\backslash x'] \star (E \mapsto e)[x\backslash x'] \star \\ \left(\begin{array}{l} P \star G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \end{array}\right)[x\backslash x'] \end{array}\right\}$$

$\Rightarrow$ { Using consequence rule, Lemma 9 with (21) and ($\star$-Monotonicity) }

$$\left\{\begin{array}{l} P \star E = (G+o) \star \\ G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star E \mapsto e \end{array}\right\} \; x := [E] \; \left\{\begin{array}{l} \exists x'.\, x = e[x\backslash x'] \star (E \mapsto e)[x\backslash x'] \star \\ \left(\begin{array}{l} P \star E = (G+o) \star \\ G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \end{array}\right)[x\backslash x'] \end{array}\right\}$$

$\Rightarrow$ { Using consequence rule and lemma for structural definition of substitution }

$$\left\{\begin{array}{l} P \star E = (G+o) \star \\ G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star E \mapsto e \end{array}\right\} \; x := [E] \; \left\{\begin{array}{l} \exists x'.\, x = e[x\backslash x'] \star \\ \left(\begin{array}{l} P \star E = (G+o) \star \\ G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star E \mapsto e \end{array}\right)[x\backslash x'] \end{array}\right\}$$

$\Rightarrow$ { Using consequence rule, and substitution with $E = G + o$ }

$$\left\{\begin{array}{l} P \star E = (G+o) \star \\ G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star (G+o) \mapsto e \end{array}\right\} \; x := [E] \; \left\{\begin{array}{l} \exists x'.\, x = e[x\backslash x'] \star \\ \left(\begin{array}{l} P \star E = (G+o) \star \\ G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star (G+o) \mapsto e \end{array}\right)[x\backslash x'] \end{array}\right\}$$

$\Rightarrow$ { Using consequence rule, Lemma 9 with (21) and ($\star$-Monotonicity) }

$$\left\{\begin{array}{l} P \star G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star (G+o) \mapsto e \end{array}\right\} \; x := [E] \; \left\{\begin{array}{l} \exists x'.\, x = e[x\backslash x'] \star \\ \left(\begin{array}{l} P \star G \mapsto \mathscr{C}_0 \star \ldots \star \\ (G+o-1) \mapsto \mathscr{C}_{o-1} \star \\ (G+o+1) \mapsto \mathscr{C}_{o+1} \star \ldots \star \\ (G+n) \mapsto \mathscr{C}_n \star \\ (G+o) \mapsto e \end{array}\right)[x\backslash x'] \end{array}\right\}$$

$\Rightarrow$ { Using consequence rule and ($\mapsto$-Group) }

$$\left\{\begin{array}{c} P \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \\ \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \end{array}\right\} \; x := [E] \; \left\{\begin{array}{l} \exists x'.\, x = e[x\backslash x'] \star \\ \left(\begin{array}{c} P \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \\ \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \end{array}\right)[x\backslash x'] \end{array}\right\}$$

$\Rightarrow$ {Using sequential composition and (22)}
$\{P \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, x := [E];\; C\, \{Q\}$

C.2 Soundness of Eval-rules

We show that the implemented unguided rule EvalUnguided can be derived from the high-level (Eval). Again for the sake of a simpler presentation we omit the context $\Pi; \Gamma$. The low-level rule is as follows:

$$\textsc{EvalUnguided}\quad \frac{\begin{array}{c}\Pi : \Upsilon \vdash_{\text{find-tr}} \exists \mathbf{u}.\, e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure} \\ \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\}\, [\mathbf{u}\backslash\mathbf{w}] \vdash_{\text{find-post}} \{\Upsilon \star R^{pure}[\mathbf{u}\backslash\mathbf{w}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\} \\ \Pi; \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i}\backslash\mathbf{v'_i}] \right\} C\, \{Q'\}\end{array}}{\Pi; \Gamma \triangleright \{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'});\ C\, \{Q'\}}\ \mathbf{v'_i}, \mathbf{w} \text{ fresh}$$

Proof. We can assume that:

(a) $\mathbf{v'_i}$ and $\mathbf{w}$ are fresh
(b) $\Upsilon \vdash_{\text{find-tr}} \exists \mathbf{x}.\, e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure}$
(c) $\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\}\, [\mathbf{u}\backslash\mathbf{w}] \vdash_{\text{find-post}} \{\Upsilon \star R^{pure}[\mathbf{x}\backslash\mathbf{w}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\}$
(d) $\Pi; \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i}\backslash\mathbf{v'_i}] \right\} C\, \{Q'\}$

By applying (Skolem) to (d) (using assumption (a)), we get:

$$\left\{ \bigvee_{i=1}^{m} \exists \mathbf{v_i}.\ \Phi_i \right\} C\, \{Q'\} \tag{23}$$

By soundness of $\vdash_{\text{find-tr}}$ and (b), we get:

$$\Upsilon \Rightarrow \exists \mathbf{u}.\, e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t'})\, \{Q\} \star R^{pure} \star P'' \tag{24}$$

for some P''.

By soundness of $\vdash_{\text{find-post}}$ and (c), we get:

$$(\forall \mathbf{a}.\, \{P\}\, k(\mathbf{t})\, \{Q\})\, [\mathbf{x}\backslash\mathbf{w}] \Rightarrow \{\Upsilon \star R^{pure}[\mathbf{x}\backslash\mathbf{w}]\}\, k(\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\} \tag{25}$$

for some fresh variable k.

To derive our low-level rule, we use the following instance of the high-level rule,

$$\frac{\Upsilon \Rightarrow e_A \mapsto \{\Upsilon\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\} \star \mathit{true}}{\{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\}}$$

We now proceed to show that the premise holds:

$$
\begin{array}{ll}
 & \Upsilon \\
\Rightarrow & \{ \text{ By (24) } \} \\
 & \exists \mathbf{u}.\, e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t'})\, \{Q\} \star R^{pure} \star P'' \\
\Rightarrow & \{\text{By (Skolem) } \} \\
 & e_A \mapsto \forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t'})\, \{Q\}\, [\mathbf{u}\backslash\mathbf{w}] \star R^{pure}[\mathbf{u}\backslash\mathbf{w}] \star P''[\mathbf{u}\backslash\mathbf{w}] \\
\Rightarrow & \{ \text{ By (25) and } (\star\text{-Monotonicity}) \ \} \\
 & e_A \mapsto \forall \mathbf{a}.\, \{\Upsilon \star R^{pure}[\mathbf{x}\backslash\mathbf{w}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\} \star R^{pure}[\mathbf{x}\backslash\mathbf{w}] \star P''[\mathbf{u}\backslash\mathbf{w}] \\
\Rightarrow & \{ \text{ By property of } \mathit{true} \text{ and } (\star\text{-Monotonicity}) \ \} \\
 & e_A \mapsto \forall \mathbf{a}.\, \{\Upsilon \star R^{pure}[\mathbf{u}\backslash\mathbf{w}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\} \star \mathit{true} \\
\Rightarrow & \{ \text{ By (Consequence), and } \Upsilon \Rightarrow \Upsilon \star R^{pure}[\mathbf{u}\backslash\mathbf{w}] \text{ as shown separately } \} \\
 & e_A \mapsto \forall \mathbf{a}.\, \{\Upsilon\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\} \star \mathit{true}
\end{array}
$$

We still need to show $\Upsilon \Rightarrow \Upsilon \star R^{pure}[\mathbf{u}\backslash\mathbf{w}]$. From (24) we can derive that $\Upsilon \Rightarrow \exists \mathbf{u}.\, R^{pure} \star \mathit{true}$ and so by skolemisation $\Upsilon \Rightarrow R^{pure}[\mathbf{u}\backslash\mathbf{w}] \star \mathit{true}$ as $\mathbf{w}$ is fresh. By ($\star$-SplitPureRight) one obtains the desired result.

We thus get by the high-level (Eval) rule instantiated as shown above:

$$
\begin{array}{ll}
 & \{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\ \Phi_i \right\} \\
\Rightarrow & \{\text{Using sequential composition (SComp) and (23)}\} \\
 & \{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'});\ C\, \{Q'\}
\end{array}
$$

C.3 Soundness of guided Eval

We now show that the Eval rule, with full hints, is sound. The rule is as follows:

$$\textsc{Eval}\quad \frac{\begin{array}{c}\Pi : \Upsilon \vdash^{\mathbf{G}}_{\text{find-tr}} \exists \mathbf{y}.\, e_A \mapsto \forall \mathbf{a}, \mathbf{b}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure} \\ (\forall \mathbf{p}.\, \{A\} \cdot (\mathbf{s})\, \{B\})\, [\mathbf{q} \backslash \mathbf{E}] \vdash_{\text{find-post}} \left\{\Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}]\right\} \cdot (\mathbf{s'}) \left\{ \bigvee_{i=1}^{n} \exists \mathbf{w}_i . \Theta_i \right\} \\ \mathit{purify}(\mathit{closure}(\Theta_i[\mathbf{w}_i \backslash \mathbf{w'_i}])) \vdash_{SMT} \mathit{false} \text{ for all } i \neq k \\ (\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\})\, [\mathbf{b} \backslash \mathbf{e}][\mathbf{y} \backslash \mathbf{y'}] \vdash_{\text{find-post}} \left\{\Theta_k[\mathbf{w}_k \backslash \mathbf{w'_k}]\right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\} \\ \Pi; \forall \mathbf{p}, \mathbf{q}.\, \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i} \backslash \mathbf{v'_i}] \right\} C\, \{Q'\} \end{array}}{\Pi; \forall \mathbf{p}, \mathbf{q}.\, \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \ \triangleright \{\, \Upsilon \,\} \quad \begin{array}{l}\mathsf{eval}\ [e_A](\mathbf{t'})\ \text{`}\mathbf{b} = \mathbf{e} \\ \quad \mathsf{before\ lookup}\ \mathbf{G} \\ \quad \mathsf{after\ lookup}\ \mathcal{L}(\mathbf{s'})\ \mathbf{q} = \mathbf{E}\ \text{'};\ C\end{array} \quad \{\, Q' \,\}}$$

$\mathbf{x'}, \mathbf{v'_i}, \mathbf{w'_i}$ fresh
$\mathcal{L}$ is a lemma procedure

Proof. We can assume that:

(a) $\mathbf{x'}, \mathbf{v'_i}, \mathbf{w'_i}$ fresh
(b) $\mathcal{L}$ is a lemma
(c) $\Pi : \Upsilon \vdash^{\mathbf{G}}_{\text{find-tr}} \exists \mathbf{x}.\, E \mapsto \forall \mathbf{a}, \mathbf{b}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure}$
(d) $(\forall \mathbf{p}.\, \{A\} \cdot (\mathbf{s})\, \{B\})\, [\mathbf{q} \backslash \mathbf{E}] \vdash_{\text{find-post}} \left\{\Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}]\right\} \cdot (\mathbf{s'}) \left\{ \bigvee_{i=1}^{n} \exists \mathbf{w}_i . \Theta_i \right\}$
(e) $\mathit{purify}(\mathit{closure}(\Theta_i[\mathbf{w}_i \backslash \mathbf{w'_i}])) \vdash_{SMT} \mathit{false}$ for all $i \neq k$
(f) $(\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\})\, [\mathbf{y} \backslash \mathbf{y'}][\mathbf{b} \backslash \mathbf{e}] \vdash_{\text{find-post}} \left\{\Theta_k[\mathbf{w}_k \backslash \mathbf{w'_k}]\right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\}$
(g) $\Pi; \forall \mathbf{p}, \mathbf{q}.\, \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i} \backslash \mathbf{v'_i}] \right\} C\, \{Q'\}$

By soundness of $\vdash_{\text{find-tr}}$ and (c), we get:

$$\Upsilon \Rightarrow \exists \mathbf{y}.\, E \mapsto \forall \mathbf{a}, \mathbf{b}.\, \{P\} \cdot (\mathbf{t})\, \{Q\} \star R^{pure} \star \Upsilon' \quad \text{for some } \Upsilon' \tag{26}$$

By soundness of $\vdash_{\text{find-post}}$ and (d), we get:

$$(\forall \mathbf{p}.\, \{A\}\, k(\mathbf{s})\, \{B\})\, [\mathbf{q} \backslash \mathbf{E}] \Rightarrow \left\{\Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}]\right\} k(\mathbf{s'}) \left\{ \bigvee_{i=1}^{n} \exists \mathbf{w}_i . \Theta_i \right\} \tag{27}$$

for a fresh k.

By Theorem 3 (soundness of $\vdash_{SMT}$) and (e), we get:

$$\mathit{purify}(\mathit{closure}(\Theta_i[\mathbf{w}_i \backslash \mathbf{w'_i}])) \Rightarrow \mathit{false} \text{ for all } i \neq k$$

which, by Lemma 11 gives:

$$\Theta_i[\mathbf{w}_i \backslash \mathbf{w'_i}] \Rightarrow \mathit{false} \text{ for all } i \neq k \tag{28}$$

By soundness of $\vdash_{\text{find-post}}$ and (f), we get:

$$(\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\})\, [\mathbf{y} \backslash \mathbf{y'}][\mathbf{b} \backslash \mathbf{e}] \Rightarrow \left\{\Theta_k[\mathbf{w}_k \backslash \mathbf{w'_k}]\right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\} \tag{29}$$

By soundness of our symbolic execution rules and (g), we get:

$$\Pi; \forall \mathbf{p}, \mathbf{q}. \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \models \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i} \backslash \mathbf{v_i'}] \right\} C \left\{ Q' \right\} \tag{30}$$

We prove this rule using the following instance of rule (EvalUnguided) already proved sound in the previous subsection.

$$\frac{\begin{array}{c} \Pi : \Upsilon \Rightarrow \exists \mathbf{y}. e_A \mapsto \forall \mathbf{a}, \mathbf{b}. \{P\} \cdot (\mathbf{t}) \{Q\} \star R^{pure} \star \Upsilon' \\ \forall \mathbf{a}, \mathbf{b}. \{P\} \cdot (\mathbf{t}) \{Q\} [\mathbf{y} \backslash \mathbf{y'}] \Rightarrow \left\{ \Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}] \right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\} \\ \Pi; \forall \mathbf{p}, \mathbf{q}. \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \models \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i} \backslash \mathbf{v_i'}] \right\} C \left\{ Q' \right\} \end{array}}{\Pi; \forall \mathbf{p}, \mathbf{q}. \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \models \{\Upsilon\}\, \mathsf{eval}\ [e_A](\mathbf{t'});\ C \left\{ Q' \right\}} \quad \mathbf{v_i'}, \mathbf{y'} \text{ fresh}$$

Because the annotations have no computational effect, the conclusions are identical.
We then must prove that the premises hold:

(a') $\Pi : \Upsilon \Rightarrow e_A \mapsto \forall \mathbf{a}, \mathbf{b}. \{P\} \cdot (\mathbf{t}) \{Q\} \star R^{pure} \star \Upsilon'$

(b') $\forall \mathbf{a}, \mathbf{b}. \{P\} \cdot (\mathbf{t}) \{Q\} [\mathbf{y} \backslash \mathbf{y'}] \Rightarrow \{\Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}]\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i . \Phi_i \right\}$

(c') $\Pi; \forall \mathbf{p}, \mathbf{q}. \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}, \Gamma \models \left\{ \bigvee_{i=1}^{m} \Phi_i[\mathbf{v_i} \backslash \mathbf{v_i'}] \right\} C \left\{ Q' \right\}$

We have (a') from (26). We have (c') from (30). This leaves us to show that (b') holds. We'll use this lemma:

Lemma 10 $\Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}] \quad \Rightarrow \quad \bigvee_{i=1}^{n} \exists \mathbf{w}_i . \Theta_i$

Proof From the procedure context of our assumptions, we have

$$\forall \mathbf{p}, \mathbf{q}. \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}$$

which, by universal instantiation is

$$\forall \mathbf{p}. \{A\}\, \mathcal{L}(\mathbf{s})\, \{B\}\, [\mathbf{q} \backslash \mathbf{E}]$$

By the fact that lemma specifications hold for skip, we know

$$\forall \mathbf{p}. \{A\}\, \mathsf{skip}\, \{B\}\, [\mathbf{q} \backslash \mathbf{E}]$$

From (27) we get,

$$\left\{ \Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}] \right\} \mathsf{skip} \left\{ \bigvee_{i=1}^{n} \exists \mathbf{w}_i . \Theta_i \right\}$$

which by the semantic fact that $\{P\}\, \mathsf{skip}\, \{Q\} \iff P \Rightarrow Q$ means that the lemma holds.

We can then prove the final premise we needed (b'):

$\forall \mathbf{a}, \mathbf{b}.\, \{P\} \cdot (\mathbf{t})\, \{Q\}\, [\mathbf{y} \backslash \mathbf{y'}]$

$\Rightarrow$ {Universal instantiation}

$\forall \mathbf{a}.\, \{P\} \cdot (\mathbf{t})\, \{Q\}\, [\mathbf{y} \backslash \mathbf{y'}][\mathbf{b} \backslash \mathbf{e}]$

$\Rightarrow$ {By (29)}

$$\left\{\Theta_k[\mathbf{w}_k \backslash \mathbf{w'_k}]\right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Phi_i \right\}$$

$\Rightarrow$ {By (ConsequenceProcedures) and (28)}

$$\left\{ \bigvee_{i=1}^{n} \Theta_i[\mathbf{w}_i \backslash \mathbf{w'_i}] \right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Phi_i \right\}$$

$\Rightarrow$ {By (Skolemize) with $\mathbf{w'_i}$ fresh}

$$\left\{ \bigvee_{i=1}^{n} \exists \mathbf{w}_i.\Theta_i \right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Phi_i \right\}$$

$\Rightarrow$ {By (ConsequenceProcedures) and Lemma 10}

$$\left\{\Upsilon \star R^{pure}[\mathbf{y} \backslash \mathbf{y'}]\right\} \cdot (\mathbf{t'}) \left\{ \bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Phi_i \right\}$$

C.4 Soundness of StoreCode

Here we prove soundness of the symbolic execution rule StoreCode (in Section B) using the core rule StoreProc (Section 4).

$$\frac{\begin{array}{c} S = (\forall \mathbf{p}|_U, \mathbf{y}.\ \{P \circ \Psi\} \cdot (\mathbf{p}|_U)\,\{Q \circ \Psi\})\,[\mathbf{p}|_{I\setminus U} \backslash \mathbf{t}|_{I\setminus U}] \\ \mathit{purify}(\Upsilon) \vdash_{SMT} e_A = e'_A + o \\ \Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \triangleright \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\,C\,\{Q'\} \end{array}}{\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \triangleright \{\,\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\,\}\ \begin{array}{l}[e_A] := \mathcal{F}(\mathbf{t}) \\ \text{`deepframe } \Psi\text{'};\ C\end{array}\ \{\,Q'\,\}}$$

where $|\mathbf{t}| = |\mathbf{p}|$ and t_i either value expression a_i or $\_$; $\mathbf{y} = \mathit{fv}(P \circ \Psi, Q \circ \Psi) - \mathbf{p}$;

$$\mathbf{p} = (p_i)_{i\in I}; \quad U = \{i \in I \mid t_i = \_\} \quad \mathbf{p}|_X = (p_i)_{i\in I\cap X}$$

So let us assume that the premises and side conditions of this rule hold. In particular for the third premise this means

$$\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \models \{\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\,C\,\{Q'\} \tag{31}$$

What we need to prove is

$$\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \models \{\,\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\,\}\ [e_A] := \mathcal{F}(\mathbf{t});\ C\ \{\,Q'\,\}$$

because the deepframe annotation has no computational effect. By the soundness of the (scomp) rule and (31) it will be enough to show

$$\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \models \begin{array}{c}\{\,\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\,\} \\ [e_A] := \mathcal{F}(\mathbf{t}) \\ \{\,\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\,\}\end{array}$$

By the second premise, Lemma 9 and Theorem 3 we have $\Upsilon \Leftrightarrow \Upsilon \star e_A = e'_A + o$ so by soundness of (consequence) the above will follow from

$$\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \models \begin{array}{c}\{\,\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n \star e_A = e'_A + o\,\} \\ [e_A] := \mathcal{F}(\mathbf{t}) \\ \{\,\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n \star e_A = e'_A + o\,\}\end{array}$$

The above will follow, by the soundness of ($\mapsto$-Group) and (ShallowFrame), from

$$\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \models \begin{array}{c}\{\,e'_A + o \mapsto \mathscr{C}_o \star e_A = e'_A + o\,\} \\ [e_A] := \mathcal{F}(\mathbf{t}) \\ \{\,e'_A + o \mapsto S \star e_A = e'_A + o\,\}\end{array}$$

By the soundness of (consequence) and (ShallowFrame), and $\Upsilon \Leftrightarrow \Upsilon \star e_A = e'_A + o$ again, the above will follow from

$$\Pi; \Gamma, \{P\}\,\mathcal{F}(\mathbf{p})\,\{Q\} \models \{e_A \mapsto \_\}\,[e_A] := \mathcal{F}(\mathbf{t})\,\{e_A \mapsto S\}$$

By the soundness of (deepframe) the above will follow from

$$\Pi; \Gamma, \{P \circ \Psi\}\,\mathcal{F}(\mathbf{p})\,\{Q \circ \Psi\} \models \{e_A \mapsto \_\}\,[e_A] := \mathcal{F}(\mathbf{t})\,\{e_A \mapsto S\} \tag{32}$$

Take the following instance of the (StoreProc) rule:

$$\{P \circ \Psi\}\,\mathcal{F}(\mathbf{p})\,\{Q \circ \Psi\} \Vdash \begin{array}{c}\{\,e_A \mapsto \_\,\} \\ [e_A] := \mathcal{F}(\mathbf{t}) \\ \{\,e_A \mapsto (\forall \mathbf{p}|_U, \mathbf{y}.\ \{P \circ \Psi\} \cdot (\mathbf{p}|_U)\,\{Q \circ \Psi\})\,[\mathbf{p}|_{I\setminus U} \backslash \mathbf{t}|_{I\setminus U}]\,\}\end{array}$$

But in fact, using the equation for S and the soundness of (StoreProc), this gives us (32) which is what we needed to prove.

C.5 Soundness of StoreCodeGuided

Here we prove soundness of the symbolic execution rule StoreCodeGuided (in Section B) using the core rule StoreProc (Section 4).

StoreCodeGuided

$$\frac{\begin{array}{c}\Pi : ((\Phi \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \;\vdash^{\emptyset}_{G^{\mathrm{pre}}}\; X(\mathbf{E}) \star \Phi' \\ \Pi : ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}] \star \mathsf{emp} \;\dashv\vdash^{\emptyset}_{G^{\mathrm{post}}}\; Y(\mathbf{e}) \star \Theta' \\ S = \left(\begin{array}{c}\{\,\exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}]\,\} \\ \forall \mathbf{p}|_U, \mathbf{y}. \qquad \cdot(\mathbf{p}|_U) \\ \{\,\exists \mathbf{w}, \mathbf{a'}.\ (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}]\,\}\end{array}\right)[\mathbf{p}|_{I\backslash U}\backslash\mathbf{t}|_{I\backslash U}] \\ \mathit{purify}(\Upsilon) \vdash_{SMT} e_A = e'_A + o \\ \Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \triangleright \{\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, C\, \{Q'\}\end{array}}{\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \triangleright \{\,\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\,\} \begin{array}{l}[e_A] := \mathcal{F}(\mathbf{t}) \\ \text{`}\mathsf{deepframe}\ \exists \mathbf{a}.\ \Upsilon' \\ \mathsf{pre}\ G^{\mathrm{pre}} \\ \mathsf{post}\ G^{\mathrm{post}}\ \text{'};\ C\end{array} \{\,Q'\,\}}$$

where t_i either value expression a_i or _;
$\mathbf{y} = \mathit{fv}(S\text{-precondition}, S\text{-postcondition}) - \mathbf{p}$; $\mathbf{v'}, \mathbf{w'}, \mathbf{a'}, \mathbf{v''}, \mathbf{w''}$ fresh
$\mathbf{p} = (p_i)_{i\in I}$; $U = \{i \in I \mid t_i = \_\}$ $\mathbf{p}|_X = (p_i)_{i\in I\cap X}$

Due to the premises of the rule we can assume that:

(a) $\Pi : ((\Phi \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \;\vdash^{\emptyset}_{G^{\mathrm{pre}}}\; X(\mathbf{E}) \star \Phi'$

(b) $\Pi : ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}] \star \mathsf{emp} \;\dashv\vdash^{\emptyset}_{G^{\mathrm{post}}}\; Y(\mathbf{e}) \star \Theta'$

(c) $S = \left(\begin{array}{c}\{\,\exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}]\,\} \\ \forall \mathbf{p}|_U, \mathbf{y}. \qquad \cdot(\mathbf{p}|_U) \\ \{\,\exists \mathbf{w}, \mathbf{a'}.\ (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}]\,\}\end{array}\right)$

(d) $\mathit{purify}(\Upsilon) \vdash_{SMT} e_A = e'_A + o$

(e) $\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \triangleright \{\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, C\, \{Q'\}$

By soundness of $\vdash^I$ and (a) we get:

$$\begin{array}{c}\Pi : ((\Phi \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \Rightarrow X(\mathbf{E}) \star \Phi' \\ \text{where } \mathit{fv}(\Phi') \subseteq \mathit{fv}(((\Phi \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}])\end{array} \tag{33}$$

By soundness of $\vdash^I$ and (b) we get:

$$\begin{array}{c}\Pi : ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}] \star \mathsf{emp} \Leftrightarrow Y(\mathbf{e}) \star \Theta' \\ \text{where } \mathit{fv}(\Theta') \subseteq \mathit{fv}(((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}] \star \mathsf{emp})\end{array} \tag{34}$$

By Theorem 3 and (d) we get:

$$\mathit{purify}(\Upsilon) \Rightarrow e_A = e'_A + o \tag{35}$$

By the soundness of our symbolic execution rules and (e) we get:

$$\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \models \{\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, C\, \{Q'\} \tag{36}$$

We now prove that this rule can be derived from the following instance of the previous unguided StoreCode:

$$\frac{\begin{array}{c}S = (\forall \mathbf{p}|_U, \mathbf{y}.\ \{\exists \mathbf{v}.\Phi \circ \exists \mathbf{a}.\Upsilon'\} \cdot (\mathbf{p}|_U)\,\{\exists \mathbf{w}.\Theta \circ \exists \mathbf{a}.\Upsilon'\})\,[\mathbf{p}|_{I\backslash U}\backslash\mathbf{t}|_{I\backslash U}] \\ \mathit{purify}(\Upsilon) \Rightarrow e_A = e'_A + o \\ \Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \models \{\Upsilon \star E' \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, C\, \{Q'\}\end{array}}{\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \models \{\,\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\,\} \begin{array}{l}[e_A] := \mathcal{F}(\mathbf{t}) \\ \text{`}\mathsf{deepframe}\ \exists \mathbf{a}.\Upsilon'\text{'};\ C\end{array} \{\,Q'\,\}}$$

where t_i either value expression a_i or _; $\mathbf{y} = \mathit{fv}(\exists \mathbf{v}.\Phi \circ \exists \mathbf{a}.\Upsilon', \exists \mathbf{w}.\Theta \circ \exists \mathbf{a}.\Upsilon') - \mathbf{p}$;

$\mathbf{p} = (p_i)_{i\in I}$; $U = \{i \in I \mid t_i = \_\}$ $\mathbf{p}|_X = (p_i)_{i\in I\cap X}$

Because the annotations have no computational effect, the conclusions are identical. Next to show the premises hold:

(a') $S = (\forall \mathbf{p}|_U, \mathbf{y}.\ \{\exists \mathbf{v}.\Phi \circ \exists \mathbf{a}.\Upsilon'\} \cdot (\mathbf{p}|_U)\,\{\exists \mathbf{w}.\Theta \circ \exists \mathbf{a}.\Upsilon'\})\,[\mathbf{p}|_{I\setminus U}\backslash \mathbf{t}|_{I\setminus U}]$
(b') $\mathit{purify}(\Upsilon) \Rightarrow e_A = e'_A + o$
(c') $\Pi; \Gamma, \{\exists \mathbf{v}.\Phi\}\,\mathcal{F}(\mathbf{p})\,\{\exists \mathbf{w}.\Theta\} \models \{\Upsilon \star e'_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, S, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\,C\,\{Q'\}$

(b') is shown by (35), and (c') is from (36). To show (a') we need to show the following implication between triples:

$$\begin{array}{ccc} \{\ \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}]\ \} & & \{\ \exists \mathbf{v}.\Phi \circ \exists \mathbf{a}.\Upsilon'\ \} \\ \cdot(\mathbf{p}|_U) & \Rightarrow & \cdot(\mathbf{p}|_U) \\ \{\ \exists \mathbf{w}, \mathbf{a'}.\ (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}]\ \} & & \{\ \exists \mathbf{w}.\Theta \circ \exists \mathbf{a}.\Upsilon'\ \} \end{array}$$

This will follow from the soundness of (ConsequenceProcedures) once we show the two premises. First we show the implication for the precondition:

$$\Pi\ :\ \exists \mathbf{v}.\ \Phi \circ (\exists \mathbf{a}.\Upsilon') \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}]$$

This is equivalent to

$$\begin{array}{ll} & \exists \mathbf{v}.\ \Phi \circ (\exists \mathbf{a}.\Upsilon') \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \\ \Leftrightarrow & \{\circ\text{-Definition}\} \\ & \exists \mathbf{v}.\ (\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star (\exists \mathbf{a}.\Upsilon') \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \\ \Leftrightarrow & \{\text{ Rename bound } \mathbf{a} \text{ by a fresh } \mathbf{a_0}\} \\ & \exists \mathbf{v}.\ (\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star (\exists \mathbf{a_0}.\Upsilon'[\mathbf{a}\backslash\mathbf{a_0}]) \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \\ \Leftrightarrow & \{\ \exists\text{-}*\text{Distribution with } \mathbf{a_0} \text{ freshness}\} \\ & \exists \mathbf{v}, \mathbf{a_0}.\ (\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a_0}] \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \end{array}$$

Using $\exists$-introduction on the left and the freshness of $\mathbf{a'}$, it suffices to prove

$$\exists \mathbf{v}.\ ((\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a_0}])[\mathbf{a_0}\backslash\mathbf{a'}] \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}]$$

Or equivalently

$$\begin{array}{ll} & \exists \mathbf{v}.\ ((\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a_0}])[\mathbf{a_0}\backslash\mathbf{a'}] \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \\ \Leftrightarrow & \{\text{ Substitution distribution with } \mathbf{a_0} \text{ freshness}\} \\ & \exists \mathbf{v}.\ (\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a_0}][\mathbf{a_0}\backslash\mathbf{a'}] \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \\ \Leftrightarrow & \{\text{ Substitution}\} \\ & \exists \mathbf{v}.\ (\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}] \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \end{array}$$

Which, by skolemization and the freshness of $\mathbf{v'}$ leaves us to prove

$$((\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \Rightarrow \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \qquad (37)$$

This is shown below:

$$
\begin{array}{ll}
 & ((\Phi \otimes (\exists \mathbf{a}.\Upsilon')) \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{v}\backslash\mathbf{v'}] \\
\Rightarrow & \{\text{By (33)}\} \\
 & X(\mathbf{E}) \star \Phi' \\
\Rightarrow & \{\text{Rename } \mathbf{v} \text{ with fresh } \mathbf{v''}\} \\
 & (X(\mathbf{E}) \star \Phi')[\mathbf{v}\backslash\mathbf{v''}] \\
\Rightarrow & \{\text{Substitution distribution with } \mathbf{v} \cap \Phi' = \emptyset \text{ (from (33))}\} \\
 & (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi') \\
\Rightarrow & \{\text{Rename } \mathbf{a'} \text{ with a fresh } \mathbf{a}_0\} \\
 & (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{a'}\backslash\mathbf{a}_0] \\
\Rightarrow & \{\exists\text{-introduction}\} \\
 & \exists \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi') \\
\Rightarrow & \{\text{Rename } \mathbf{v'} \text{ with } \mathbf{v} \ (\mathbf{v} \cap \Phi' = \emptyset \text{ (from (33)))}\} \\
 & \exists \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}] \\
\Rightarrow & \{\text{Rename } \mathbf{v} \text{ with a fresh } \mathbf{v}_0\} \\
 & \exists \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}][\mathbf{v}\backslash\mathbf{v}_0] \\
\Rightarrow & \{\exists\text{-introduction}\} \\
 & \exists \mathbf{v}, \mathbf{a'}.\ (X(\mathbf{E})[\mathbf{v}\backslash\mathbf{v''}] \star \Phi')[\mathbf{v'}\backslash\mathbf{v}]
\end{array}
$$

Next, we show the entailment for the postcondition

$$\Pi \ : \ \exists \mathbf{w}, \mathbf{a'}.\ (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}] \Rightarrow \exists \mathbf{w}.\Theta \circ (\exists \mathbf{a}.\Upsilon')$$

By skolemization and choosing a fresh $\mathbf{a}_1, \mathbf{w}_1$ leaves us to prove

$$(Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}][\mathbf{a'}\backslash\mathbf{a}_1][\mathbf{w}\backslash\mathbf{w}_1] \Rightarrow \exists \mathbf{w}.\Theta \circ (\exists \mathbf{a}.\Upsilon') \tag{38}$$

This is proved below

$$
\begin{array}{ll}
 & (Y(\mathbf{e})[\mathbf{w}\backslash\mathbf{w''}] \star \Theta')[\mathbf{w'}\backslash\mathbf{w}][\mathbf{a'}\backslash\mathbf{a}_1][\mathbf{w}\backslash\mathbf{w}_1] \\
\Rightarrow & \{\ \text{Substitution distribution with } \mathbf{w} \cap \Theta' = \emptyset \text{ (from (34))}\ \} \\
 & (Y(\mathbf{e}) \star \Theta')[\mathbf{w}\backslash\mathbf{w''}][\mathbf{w'}\backslash\mathbf{w}][\mathbf{a'}\backslash\mathbf{a}_1][\mathbf{w}\backslash\mathbf{w}_1] \\
\Rightarrow & \{\ \text{from (34)}\ \} \\
 & ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}][\mathbf{w}\backslash\mathbf{w''}][\mathbf{w'}\backslash\mathbf{w}][\mathbf{a'}\backslash\mathbf{a}_1][\mathbf{w}\backslash\mathbf{w}_1] \\
\Rightarrow & \{\ \text{remove redundant substitution } [\mathbf{w}\backslash\mathbf{w''}] \text{ (all } \mathbf{w} \text{ have been substituted)}\} \\
 & ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a'}])[\mathbf{w}\backslash\mathbf{w'}][\mathbf{w'}\backslash\mathbf{w}][\mathbf{a'}\backslash\mathbf{a}_1][\mathbf{w}\backslash\mathbf{w}_1] \\
\Rightarrow & \{\ \text{apply substitution } [\mathbf{a'}\backslash\mathbf{a}_1] \text{ with } \mathbf{a'} \text{ freshness (so not already appearing)}\ \} \\
 & ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a}_1])[\mathbf{w}\backslash\mathbf{w'}][\mathbf{w'}\backslash\mathbf{w}][\mathbf{w}\backslash\mathbf{w}_1] \\
\Rightarrow & \{\ \text{apply substitutions } [\mathbf{w}\backslash\mathbf{w'}][\mathbf{w'}\backslash\mathbf{w}] \text{ with } \mathbf{w'} \text{ freshness (so not already appearing)}\ \} \\
 & ((\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a}_1])[\mathbf{w}\backslash\mathbf{w}_1] \\
\Rightarrow & \{\ \exists\text{-introduction}\ \} \\
 & \exists \mathbf{w}.\ (\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \Upsilon'[\mathbf{a}\backslash\mathbf{a}_1] \\
\Rightarrow & \{\ \exists\text{-introduction with } \star\text{-}\textsc{Monotonicity}\ \} \\
 & \exists \mathbf{w}.\ (\Theta \otimes \exists \mathbf{a}.\Upsilon') \star \exists \mathbf{a}.\Upsilon' \\
\Rightarrow & \{\circ\text{-}\textsc{Definition}\} \\
 & \exists \mathbf{w}.\ \Theta \circ (\exists \mathbf{a}.\Upsilon')
\end{array}
$$

Thus, by applying ConsequenceProcedures to (c) with (37) and (38), we get (a').

D Proofs of soundness of rules for entailments between disjuncts

The general pattern for proving a rule

$$\frac{\Phi' \ \vdash^{I'} \ \exists \mathbf{v}' \ . \ \Psi' \star \Theta'}{\Phi \ \vdash^{I} \ \exists \mathbf{v} \ . \ \Psi \star \Theta} S$$

(where the side conditions S depend only on $\Phi, \mathbf{v}, \Psi$) is as follows.

1. Assume that $fv(\Phi) \cap \mathbf{v} = \emptyset$ and assume that the side conditions S hold.
2. Prove that $fv(\Phi') \cap \mathbf{v}' = \emptyset$.
3. Assume further that:
   (a') $\Phi' \ \Rightarrow \ \Psi'[\mathbf{v}' \backslash I'(\mathbf{v}')] \star \Theta'$
   (b') $fv(\Theta') \subseteq fv(\Phi')$
   (c') $dom(I') = \mathbf{v}'$
4. Prove that:
   (a) $\Phi \ \Rightarrow \ \Psi[\mathbf{v} \backslash I(\mathbf{v})] \star \Theta$
   (b) $fv(\Theta) \subseteq fv(\Phi)$
   (c) $dom(I) = \mathbf{v}$

We include three such proofs, for the rules InstUsingEq, CancelPt1 and InstMatchAddr. We begin with InstUsingEq.

$$\frac{\begin{array}{c}\textsc{InstUsingEq} \\ \Phi \ \vdash^{I} \ \exists \mathbf{v} \ . \ \Psi[v \backslash e] \star \Theta \end{array}}{\Phi \ \vdash^{I[v:=e]} \ \exists \mathbf{v}, v \ . \ \Psi \star v = e \star \Theta} \quad fv(e) \cap \mathbf{v}, v = \emptyset$$

Proof As per 1. we can assume that $fv(\Phi) \cap \mathbf{v}, v = \emptyset$ and that $fv(e) \cap \mathbf{v}, v = \emptyset$. For 2. we need to prove $fv(\Phi) \cap \mathbf{v} = \emptyset$, which follows easily from our assumptions. As per 3. we now assume further that:

(a') $\Phi \ \Rightarrow \ \Psi[v \backslash e][\mathbf{v} \backslash I(\mathbf{v})] \star \Theta$
(b') $fv(\Theta) \subseteq fv(\Phi)$
(c') $dom(I) = \mathbf{v}$

and for 4. we need to prove:

(a) $\Phi \ \Rightarrow \ (\Psi \star v = e)[\mathbf{v}, v \backslash I[v := e](\mathbf{v}, v)] \star \Theta$
(b) $fv(\Theta) \subseteq fv(\Phi)$
(c) $dom(I[v := e]) = \mathbf{v}, v$

(b) is exactly (b'). (c) follows from (c'). Now to show (a). The following argument shows that (a) and (a') are equivalent.

$$\begin{array}{ll} & (\Psi \star v = e)[\mathbf{v}, v \backslash I[v := e](\mathbf{v}, v)] \star \Theta \\ \Leftrightarrow & \{\text{distribute substitution over } \star\} \\ & \Psi[\mathbf{v}, v \backslash I[v := e](\mathbf{v}, v)] \star (v = e)[\mathbf{v}, v \backslash I[v := e](\mathbf{v}, v)] \star \Theta \\ \Leftrightarrow & \{\text{because } fv(e) \cap \mathbf{v} = \emptyset\} \\ & \Psi[v \backslash e][\mathbf{v} \backslash I(\mathbf{v})] \star (v = e)[\mathbf{v}, v \backslash I[v := e](\mathbf{v}, v)] \star \Theta \\ \Leftrightarrow & \{\text{because } fv(e) \cap \mathbf{v}, v = \emptyset\} \\ & \Psi[v \backslash e][\mathbf{v} \backslash I(\mathbf{v})] \star e = e \star \Theta \\ \Leftrightarrow & \{\text{because } e = e \Leftrightarrow \mathsf{emp} \text{ - where does that come from?}\} \\ & \Psi[v \backslash e][\mathbf{v} \backslash I(\mathbf{v})] \star \Theta \end{array}$$

Next we tackle the CancelPt1 rule.

$$\textsc{CancelPt1}\quad \frac{\Phi \ \vdash^I \ \exists \mathbf{v} \,.\, \Psi \star \Theta}{\Phi \star e_A \mapsto \mathscr{C} \ \vdash^I \ \exists \mathbf{v} \,.\, \Psi \star e'_A \mapsto \_ \star \Theta} \quad \begin{array}{l} fv(e') \cap \mathbf{v} = \emptyset, \\ purify(\Phi) \vdash_{SMT} e = e' \end{array}$$

Proof As per 1. we can assume that $fv(\Phi \star e \mapsto \mathscr{C}) \cap \mathbf{v} = \emptyset$, $fv(e') \cap \mathbf{v} = \emptyset$, and $purify(\Phi) \vdash_{SMT} e_A = e'_A$. For 2. we need to prove $fv(\Phi) \cap \mathbf{v} = \emptyset$, which follows easily from our assumptions. As per 3. we now assume further that:

(a') $\Phi \ \Rightarrow \ \Psi[\mathbf{v} \backslash I(\mathbf{v})] \star \Theta$
(b') $fv(\Theta) \subseteq fv(\Phi)$
(c') $dom(I) = \mathbf{v}$

and for 4. we need to prove:

(a) $\Phi \star e_A \mapsto \mathscr{C} \ \Rightarrow \ (\Psi \star e'_A \mapsto \_)[\mathbf{v} \backslash I(\mathbf{v})] \star \Theta$
(b) $fv(\Theta) \subseteq fv(\Phi \star e_A \mapsto \mathscr{C})$
(c) $dom(I) = \mathbf{v}$

(b) follows easily from (b'). (c) is exactly (c'). Now to show (a).

$$\begin{array}{ll} & \Phi \star e_A \mapsto \mathscr{C} \\ \Rightarrow & \{\text{using } purify(\Phi) \vdash_{SMT} e_A = e'_A \text{ and what else?}\} \\ & \Phi \star e'_A \mapsto \mathscr{C} \\ \Rightarrow & \{\text{using } \star\text{-MONOTONICITY to add } e'_A \mapsto \mathscr{C} \text{ to both sides of (a')}\} \\ & \Psi[\mathbf{v} \backslash I(\mathbf{v})] \star e'_A \mapsto \mathscr{C} \star \Theta \\ \Rightarrow & \{\text{using } \star\text{-MONOTONICITY and } e'_A \mapsto \mathscr{C} \Rightarrow e' \mapsto \_\} \\ & \Psi[\mathbf{v} \backslash I(\mathbf{v})] \star e'_A \mapsto \_ \star \Theta \\ \Rightarrow & \{\text{since } fv(e'_A) \cap \mathbf{v} = \emptyset\} \\ & \Psi[\mathbf{v} \backslash I(\mathbf{v})] \star (e'_A \mapsto \_)[\mathbf{v} \backslash I(\mathbf{v})] \star \Theta \\ \Rightarrow & \{\text{since } fv(e'_A) \cap \mathbf{v} = \emptyset\} \\ & \Psi[\mathbf{v} \backslash I(\mathbf{v})] \star (e'_A \mapsto \_)[\mathbf{v} \backslash I(\mathbf{v})] \star \Theta \\ \Rightarrow & \{\text{using distribution of substitution and } \star\} \\ & (\Psi \star e'_A \mapsto \_)[\mathbf{v} \backslash I(\mathbf{v})] \star \Theta \end{array}$$

Next we prove the InstMatchAddr rule.

$$\textsc{InstMatchAddr}\quad \frac{\Phi \star e_A \mapsto \mathscr{C} \ \vdash^I \ \exists \mathbf{v} \,.\, (\Psi \star v \mapsto \mathscr{C}')[v \backslash e_A] \star \Theta}{\Phi \star e_A \mapsto \mathscr{C} \ \vdash^{I[v:=e_A]} \ \exists \mathbf{v}, v \,.\, \Psi \star v \mapsto \mathscr{C}' \star \Theta} \quad \textbf{backtracks}$$

Proof As per 1. we can assume that $fv(\Phi \star e_A \mapsto \mathscr{C}) \cap \mathbf{v}, v = \emptyset$. For 2. we need to prove $fv(\Phi \star e_A \mapsto \mathscr{C}) \cap \mathbf{v} = \emptyset$, which follows easily.

As per 3. we now assume further that:

(a') $\Phi \star e_A \mapsto \mathscr{C} \ \Rightarrow \ (\Psi \star v \mapsto \mathscr{C}')[v \backslash e_A][\mathbf{v} \backslash I(\mathbf{v})] \star \Theta$
(b') $fv(\Theta) \subseteq fv(\Phi \star e_A \mapsto \mathscr{C})$
(c') $dom(I) = \mathbf{v}$

and for 4. we need to prove:

(a) $\Phi \star e_A \mapsto \mathscr{C} \ \Rightarrow \ (\Psi \star v \mapsto \mathscr{C}')[\mathbf{v}, v \backslash I[v := e_A](\mathbf{v}, v)] \star \Theta$

(b) $fv(\Theta) \subseteq fv(\Phi \star e \mapsto \mathscr{C})$
(c) $dom(I[v := e_A]) = \mathbf{v}, v$

(b) is exactly (b') and (c) follows easily from (c'). The following argument shows that (a) and (a') are equivalent.

$$
\begin{array}{ll}
 & (\Psi \star v \mapsto \mathscr{C}')[v\backslash e_A][\mathbf{v}\backslash I(\mathbf{v})] \star \Theta \\
\Leftrightarrow & \{ \text{ since } fv(e_A) \cap \mathbf{v} = \emptyset \text{ which follows from } fv(\Phi \star e_A \mapsto \mathscr{C}) \cap \mathbf{v}, v = \emptyset \ \} \\
 & (\Psi \star v \mapsto \mathscr{C}')[\mathbf{v}, v\backslash I[v := e_A](\mathbf{v}, v)] \star \Theta
\end{array}
$$

FoldPredRight

$$
\frac{\begin{array}{c}(\mathrm{X}(\mathbf{x}) \Leftrightarrow (\exists \mathbf{v}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \mathbf{v}_n.\ \Upsilon_n)) \ \in \ \Pi \\ \Upsilon_1[\mathbf{x}\backslash\mathbf{e}][\mathbf{v_1}\backslash\mathbf{d_1}] \star \Theta \ \vdash^{\emptyset} \ \Phi \star \mathsf{emp} \qquad \ldots \qquad \Upsilon_n[\mathbf{x}\backslash\mathbf{e}][\mathbf{v_n}\backslash\mathbf{d_n}] \star \Theta \ \vdash^{\emptyset} \ \Phi \star \mathsf{emp}\end{array}}{\Pi : X(\mathbf{e}) \star \Theta \ \vdash^{\emptyset}_{G} \ \Phi \star \mathsf{emp}}
$$

$$
\begin{array}{c}\mathbf{d_1} \ldots \mathbf{d_n} \text{ fresh} \\ G = \mathsf{fold}\ X(\_)\ \_\end{array}
$$

Proof We have no existential variables, so 2. comes for free. As per 3. we now assume further that:

(a') $\Upsilon_i[\mathbf{x}\backslash\mathbf{e}][\mathbf{v_i}\backslash\mathbf{d_i}] \star \Theta \ \Rightarrow \ \Phi \star \mathsf{emp}$ forall $i \in \{1, \ldots, n\}$
(b') $fv(\mathsf{emp}) \subseteq fv(\Upsilon_1[\mathbf{x}\backslash\mathbf{e}][\mathbf{v_1}\backslash\mathbf{d_1}] \star \Theta)$
(c') $dom(\emptyset) = \emptyset$

and for 4. we need to prove:

(a) $X(\mathbf{e}) \star \Theta \ \Rightarrow \ \Phi \star \mathsf{emp}$
(b) $fv(\mathsf{emp}) \subseteq fv(X(\mathbf{e}) \star \Theta)$
(c) $dom(\emptyset) = \emptyset$

(b) and (c) holds trivially.

By skolemization from (a') and the fact that $\mathbf{d}_1, \ldots, \mathbf{d}_n$ are fresh, and that $fv(\Theta) \cap \mathbf{v_i} = \emptyset$, we get:

$$\exists \mathbf{v_i}.\ \Upsilon_i[\mathbf{x}\backslash\mathbf{e}] \star \Theta \ \Rightarrow \ \Phi \star \mathsf{emp} \quad \text{forall } i \in \{1, \ldots, n\} \tag{39}$$

The following argument proves (a)

$$
\begin{array}{ll}
 & X(\mathbf{e}) \star \Theta \\
\Rightarrow & \{\text{PredRight with premise}\} \\
 & ((\exists \mathbf{v}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \mathbf{v}_n.\ \Upsilon_n))[\mathbf{x}\backslash\mathbf{e}] \star \Theta \\
\Rightarrow & \{\text{Substitution distribution}\} \\
 & ((\exists \mathbf{v}_1.\ \Upsilon_1[\mathbf{x}\backslash\mathbf{e}]) \vee \ldots \vee (\exists \mathbf{v}_n.\ \Upsilon_n[\mathbf{x}\backslash\mathbf{e}])) \star \Theta \\
\Rightarrow & \{\star\text{-distribution}\} \\
 & (\exists \mathbf{v}_1.\ \Upsilon_1[\mathbf{x}\backslash\mathbf{e}] \star \Theta) \vee \ldots \vee (\exists \mathbf{v}_n.\ \Upsilon_n[\mathbf{x}\backslash\mathbf{e}] \star \Theta) \\
\Rightarrow & \{\text{Rewrite}\} \\
 & \exists \mathbf{v}_i.\ \Upsilon_i[\mathbf{x}\backslash\mathbf{e}] \star \Theta \quad \text{forall } i \in \{1, \ldots, n\} \\
\Rightarrow & \{\text{By (39)}\} \\
 & \Phi \star \mathsf{emp}
\end{array}
$$

Next we'll do the FoldPredLeft rule.

$$\textsc{FoldPredLeft}\quad \frac{\mathrm{X}(\mathbf{v}) \Leftrightarrow (\exists \mathbf{v}_1, \mathbf{b}_1.\ \Upsilon_1) \vee \ldots \vee (\exists \mathbf{v}_n, \mathbf{b}_n.\ \Upsilon_n) \ \in\ \Pi \quad \Phi \ \vdash^I\ \exists \mathbf{w}, \mathbf{c}.\ \Upsilon_i[\mathbf{v}\backslash\mathbf{q}][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}] \star \Theta}{\Pi : \Phi \ \vdash^{\emptyset}_{G}\ X(\mathbf{e}) \star \Theta}$$

1. $G = \mathsf{fold}\ X(\mathbf{p})\ \mathbf{b} = \mathbf{y}$,
2. $\mathbf{b_i} \subseteq \mathbf{b}$, 3. $\mathbf{b} - \mathbf{b_i} \cap (fv(\Upsilon_i) \cup \mathbf{p}) = \emptyset$
4. $1 \leq i \leq n$, 5. $\mathbf{c}, \mathbf{w}$ fresh,
6. each p_i is an expression or ?,
7. each q_i is $\begin{cases} c_i & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}$,
8. each e_i is $\begin{cases} I(c_i) & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}$

Proof We have no existential variables in the conclusion, so we don't need assumption 1. For 2. we need to prove $fv(\Phi) \cap (\mathbf{w}, \mathbf{c}) = \emptyset$, which follows from side-condition (5), the freshness of $\mathbf{w}$

As per 3. we now assume further that:

(a') $\Phi \ \Rightarrow\ \Upsilon_i[\mathbf{x}\backslash\mathbf{q}][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}][\mathbf{w}\backslash I(\mathbf{w})][\mathbf{c}\backslash I(\mathbf{c})] \star \Theta$
(b') $fv(\Theta) \subseteq fv(\Phi)$
(c') $dom(I) = \mathbf{w}, \mathbf{c}$

and for 4. we need to prove:

(a) $\Phi \ \Rightarrow\ X(\mathbf{e}) \star \Theta$
(b) $fv(\Theta) \subseteq fv(\Phi)$
(c) $dom(\emptyset) = \emptyset$

(b) is exactly (b'), and (c) holds trivially. The following argument shows (a)

$$\begin{array}{ll} & \Phi \\ \Rightarrow & \{\ \text{From (a')}\ \} \\ & \Upsilon_i[\mathbf{x}\backslash\mathbf{q}][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}][\mathbf{w}\backslash I(\mathbf{w})][\mathbf{c}\backslash I(\mathbf{c})] \star \Theta \\ \Rightarrow & \{\ \text{From side-condition 7 } (\mathbf{q} \text{ is } \mathbf{c}_i \text{ or } \mathbf{p}_i)\ \} \\ & \Upsilon_i[\mathbf{x}\backslash(\mathbf{c}|\mathbf{p})][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}][\mathbf{w}\backslash I(\mathbf{w})][\mathbf{c}\backslash I(\mathbf{c})] \star \Theta \\ \Rightarrow & \{\ \text{Substitution with freshness of } \mathbf{c} \text{ (so not in } fv(\Upsilon_i))\ \} \\ & \Upsilon_i[\mathbf{x}\backslash(I(\mathbf{c})|\mathbf{p})][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}][\mathbf{w}\backslash I(\mathbf{w})] \star \Theta \\ \Rightarrow & \{\ \text{From side-condition 8 } (\mathbf{e} \text{ is } I(\mathbf{c}_i) \text{ or } \mathbf{p}_i)\ \} \\ & \Upsilon_i[\mathbf{x}\backslash\mathbf{e}][\mathbf{b}\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}][\mathbf{w}\backslash I(\mathbf{w})] \star \Theta \\ \Rightarrow & \{\ \text{From side-condition 2 and 3}\ \} \\ & \Upsilon_i[\mathbf{x}\backslash\mathbf{e}][\mathbf{b}_i\backslash\mathbf{y}][\mathbf{v}_i\backslash\mathbf{w}][\mathbf{w}\backslash I(\mathbf{w})] \star \Theta \\ \Rightarrow & \{\ \text{Substitution with freshness of } \mathbf{w} \text{ (so not in } fv(\Upsilon_i))\ \} \\ & \Upsilon_i[\mathbf{x}\backslash\mathbf{e}][\mathbf{b}_i\backslash\mathbf{y}][\mathbf{v}_i\backslash I(\mathbf{w})] \star \Theta \\ \Rightarrow & \{\ \exists\text{-introduction}\ \} \\ & (\exists \mathbf{b}_i.\ \Upsilon_i[\mathbf{x}\backslash\mathbf{e}][\mathbf{v}_i\backslash I(\mathbf{w})]) \star \Theta \\ \Rightarrow & \{\ \exists\text{-introduction}\ \} \\ & (\exists \mathbf{v}_i, \mathbf{b}_i.\ \Upsilon_i[\mathbf{x}\backslash\mathbf{e}]) \star \Theta \\ \Rightarrow & \{\ \text{Premise}\ \} \\ & X(\mathbf{x})[\mathbf{x}\backslash\mathbf{e}] \star \Theta \\ \Rightarrow & \{\ \text{Substitution}\ \} \\ & X(\mathbf{e}) \star \Theta \end{array}$$

E Proof search algorithm for entailments between specifications

Currently we have four rules for the judgement $B_1 \vdash B_2$, which are applied in the order we present them. The first rule removes all the quantifiers on the RHS:

$$\textsc{Remove}\forall\textsc{Right}\ \frac{\forall \mathbf{y}.B \ \vdash\ B'[\mathbf{y}'\backslash\mathbf{z}]}{\forall \mathbf{y}.B \ \vdash\ \forall \mathbf{y}'.B'}\ \ \mathbf{z}\text{ fresh}$$

Soundness: The premise gives us $\forall \mathbf{y}.B \ \Rightarrow\ B'[\mathbf{y}'\backslash\mathbf{z}]$ from which, using universal generalisation, we get $\forall \mathbf{z}.(\forall \mathbf{y}.B \ \Rightarrow\ B'[\mathbf{y}'\backslash\mathbf{z}])$. Using (41) we then derive $\forall \mathbf{z}.\forall \mathbf{x}.B \ \Rightarrow\ \forall \mathbf{z}.B'[\mathbf{y}'\backslash\mathbf{z}]$. But $\mathbf{z}$ don't appear in B, so we have just $\forall \mathbf{y}.B \ \Rightarrow\ \forall \mathbf{z}.B'[\mathbf{y}'\backslash\mathbf{z}]$. Finally we rename the bound variables $\mathbf{z}$ to $\mathbf{y}'$ giving $\forall \mathbf{y}.B \ \Rightarrow\ \forall \mathbf{y}'.B'$ as required.

The second rule deals with disjunctions in the precondition of the triple on the right.

$$\textsc{DisjPre}\ \frac{\bigwedge_{i=1}^{n} \forall \mathbf{y}.B \ \vdash\ \{\exists \mathbf{v}_i.A_i\}\cdot(\mathbf{t})\,\{Q\}}{\forall \mathbf{y}.B \ \vdash\ \{\exists \mathbf{v}_1.A_1 \vee \ldots \vee \exists \mathbf{v}_n.A_n\}\cdot(\mathbf{t})\,\{Q\}}$$

Soundness: By iterated application of Disjunction.

The third rule uses Skolemisation to deal with existential quantifiers in the precondition of the specification on the right.

$$\textsc{ExistsPre}\ \frac{\forall \mathbf{y}.B \ \vdash\ \{A[\mathbf{v}\backslash\mathbf{a}]\}\cdot(\mathbf{t})\,\{Q\}}{\forall \mathbf{y}.B \ \vdash\ \{\exists \mathbf{v}.A\}\cdot(\mathbf{t})\,\{Q\}}\ \mathbf{a}\text{ fresh}$$

Soundness: By (Skolem).

The final rule uses the prover for $\vdash_{\text{find-post}}$ to do most of the work (here A' is just a spatial conjunction but A, B can be any assertions):

$$\textsc{TripleEnt}\ \frac{B \ \vdash_{\text{find-post}}\ \{\Phi\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon_i\right\} \qquad \bigwedge_{i=1}^{m} \Upsilon_i[\mathbf{v}_i\backslash\mathbf{a}_i] \ \vdash^{I_i}\ \exists \mathbf{b}_{j_i}.(\Upsilon'_{j_i}[\mathbf{w}_{j_i}\backslash\mathbf{b}_{j_i}]) \star \Theta_i^{pure}}{B \ \vdash\ \{\Phi\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m'} \exists \mathbf{w}_i.\Upsilon'_i\right\}}$$

1. $j_1, \ldots, j_m \in \{1, \ldots, m'\}$,
2. $\mathbf{a}_1, \ldots, \mathbf{a}_m$ all chosen fresh,
3. $\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_{m'}}$ all chosen fresh
4. $\Theta_1, \ldots, \Theta_m$ pure

Soundness: We get from the first premise and the soundness of the proof system for $\vdash_{\text{find-post}}$ the following:

$$B \ \Rightarrow\ \{\Phi\}\cdot(\mathbf{t})\left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon_i\right\}$$

We now need to check the well-formedness of the second premise, that is, we need to check that for all $i \in \{1, \ldots, m\}$ we have $fv(\Upsilon_i[\mathbf{v}_i\backslash\mathbf{a}_i]) \cap \mathbf{b}_{j_i} = \emptyset$. This is immediate from side condition 3.

Now, we can use the consequence rule to obtain our desired conclusion if we can show

$$\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon_i \ \Rightarrow\ \bigvee_{i=1}^{m'} \exists \mathbf{w}_i.\Upsilon'_i$$

This will follow if we can show that for all $i \in \{1, \ldots, m\}$,

$$\exists \mathbf{v}_i.\Upsilon_i \;\Rightarrow\; \exists \mathbf{w}_{j_i}.\Upsilon'_{j_i}$$

The above is equivalent to

$$\exists \mathbf{v}_i.\Upsilon_i \;\Rightarrow\; \exists \mathbf{b}_{j_i}.(\Upsilon'_{j_i}[\mathbf{w}_{j_i} \backslash \mathbf{b}_{j_i}])$$

because the variables $\mathbf{b}_{j_i}$ are chosen fresh. Using skolemisation, and the fact that variables $\mathbf{a}_i$ are chosen fresh, it will be enough to show

$$\Upsilon_i[\mathbf{v}_i \backslash \mathbf{a}_i] \;\Rightarrow\; \exists \mathbf{b}_{j_i}.(\Upsilon'_{j_i}[\mathbf{w}_{j_i} \backslash \mathbf{b}_{j_i}])$$

The following argument shows this:

$$\begin{array}{ll} & \Upsilon_i[\mathbf{v}_i \backslash \mathbf{a}_i] \\ \Rightarrow & \{\text{using the rule's second premise}\} \\ & \exists \mathbf{b}_{j_i}.(\Upsilon'_{j_i}[\mathbf{w}_{j_i} \backslash \mathbf{b}_{j_i}]) \star \Theta_i^{pure} \\ \Rightarrow & \{\text{by SplitPureLeft}\} \\ & \exists \mathbf{b}_{j_i}.(\Upsilon'_{j_i}[\mathbf{w}_{j_i} \backslash \mathbf{b}_{j_i}]) \end{array}$$

F Proof search algorithm for $\vdash_{\text{find-post}}$

The first rule we have for $\vdash_{\text{find-post}}$ instantiates quantifiers on the LHS so that the parameters in the two specifications match.

InstParam

$$\frac{(\forall \mathbf{a}\ .\ \{P\} \cdot (\mathbf{t_1}, y, \mathbf{t_2})\, \{Q\})\, [y \backslash t] \quad \vdash_{\text{find-post}} \quad \{\Phi\} \cdot (\mathbf{t_1}, t, \mathbf{t_2})\, \{Q'\}}{\forall \mathbf{a}, y\ .\ \{P\} \cdot (\mathbf{t_1}, y, \mathbf{t_2})\, \{Q\} \quad \vdash_{\text{find-post}} \quad \{\Phi\} \cdot (\mathbf{t_1}, t, \mathbf{t_2})\, \{Q'\}} \quad \mathbf{t_1} \cap (\mathbf{a} \cup \{y\}) = \emptyset$$

(The side condition here makes sure that we instantiate the *leftmost* quantified parameter, to make things more deterministic.)

Soundness: Follows from soundness of universal instantiation, that is, $\forall \mathbf{u}, y.A \;\Rightarrow\; \forall.\mathbf{u}(A[y \backslash e])$ for any expression e.

The main rule for $\vdash_{\text{find-post}}$ is the following one.

InferSpecForCall

$$\frac{\Phi \;\vdash^I\; \exists \mathbf{u}_k, \mathbf{a}.\Upsilon_k \star \Theta}{\forall \mathbf{a}.\left\{\bigvee_{i=1}^{n} \exists \mathbf{u}_i.\Upsilon_i\right\} \cdot (\mathbf{p}) \left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right\} \;\vdash_{\text{find-post}}\; \{\Phi\} \cdot (\mathbf{t}) \left\{\bigvee_{i=1}^{m} (\exists \mathbf{v}_i.\Upsilon'_i[\mathbf{a} \backslash I(\mathbf{a})] \star \Theta)\right\}}$$

1. $\mathbf{t} \cap \mathbf{a} = \emptyset$,
2. $fv(\Phi) \cap \mathbf{u}_k, \mathbf{a} = \emptyset$,
3. for each $i \in \{1, \ldots, m\}$ we have $\mathbf{v}_i \cap \mathbf{a} = \emptyset$
4. for each $i \in \{1, \ldots, m\}$, no formula in $I(\mathbf{a})$ contains a variable from $\mathbf{v}_i$
5. $k \in \{1, \ldots, n\}$
6. no formula in $I(\mathbf{u}_k)$ contains a variable from $\mathbf{a}$
7. $\mathbf{u}_k \cap \mathbf{a} = \emptyset$

backtracks

Note that after (InstParam) has been used to make the parameters match, some bound variables may need to be renamed to fresh variables to allow the (InferSpecForCall) rule to be used. We do not go into detail about these renamings.

Note also that, when called from (TripleEnt), this rule needs to backtrack in its choice of k. If one choice of k fails, it is worth backtracking *even if one had shown* $\Phi \;\vdash^I\; \exists \mathbf{u}_k, \mathbf{x}.\Upsilon_k \star \Theta$, because different choices of k might succeed with a different I, giving different instantiations of the universally quantified variables $\mathbf{a}$ and thus different postconditions.

Soundness: For now we will assume that the following holds:

$$\Upsilon_k[\mathbf{u}_k, \mathbf{a} \backslash I(\mathbf{u}_k, \mathbf{a})] \;\Rightarrow\; (\exists \mathbf{u}_k.\Upsilon_k)\, [\mathbf{a} \backslash I(\mathbf{a})] \tag{40}$$

We will come back and prove this at the end.

First we need to show that the entailment problem in the first premise is "well-formed", which means showing $fv(\Phi) \cap \mathbf{u}_i, \mathbf{a} = \emptyset$. But this is just side condition 2. Then, we get to assume that:

(a) $\Phi \;\Rightarrow\; \Upsilon_i[\mathbf{u}_i, \mathbf{a}\backslash I(\mathbf{u}_i, \mathbf{a})] \star \Theta$
(b) $fv(\Theta) \subseteq fv(\Phi)$
(c) $dom(I) = \mathbf{u}_i, \mathbf{a}$

We'll need to use the following axiom scheme (which one finds in Hilbert systems):

$$\forall \mathbf{x}.(A \Rightarrow B) \;\Rightarrow\; (\forall \mathbf{x}.A \Rightarrow \forall \mathbf{x}.B) \tag{41}$$

We then reason as follows.

$$\begin{array}{ll}
 & \forall \mathbf{a}.\left\{\bigvee_{i=1}^{n} \exists \mathbf{u}_i.\Upsilon_i\right\} \cdot (\mathbf{t})\left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right\} \\
\Rightarrow & \{\text{consequence rule, universal generalisation and (41)}\} \\
 & \forall \mathbf{a}.\{\Upsilon_k\} \cdot (\mathbf{t})\left\{\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right\} \\
\Rightarrow & \{\text{instantiate } \mathbf{a} \text{ with } I(\mathbf{a}) \text{ and use side condition 1}\} \\
 & \{(\exists \mathbf{u}_k.\Upsilon_k)\,[\mathbf{a}\backslash I(\mathbf{a})]\} \cdot (\mathbf{t})\left\{\left(\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon_i\right)[\mathbf{a}\backslash I(\mathbf{a})]\right\} \\
\Rightarrow & \{\text{use shallow frame axiom to add } \Theta \text{ as a frame}\} \\
 & \{(\exists \mathbf{u}_k.\Upsilon_k)\,[\mathbf{a}\backslash I(\mathbf{a})] \star \Theta\} \cdot (\mathbf{t})\left\{\left(\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right)[\mathbf{a}\backslash I(\mathbf{a})] \star \Theta\right\} \\
\Rightarrow & \{\text{use consequence rule, (40) and monotonicity of } \star\} \\
 & \{\Upsilon_k[\mathbf{u}_k, \mathbf{a}\backslash I(\mathbf{u}_k, \mathbf{a})] \star \Theta\} \cdot (\mathbf{t})\left\{\left(\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right)[\mathbf{a}\backslash I(\mathbf{a})] \star \Theta\right\} \\
\Rightarrow & \{\text{using consequence rule and (a) above}\} \\
 & \{\Phi\} \cdot (\mathbf{t})\left\{\left(\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i\right)[\mathbf{a}\backslash I(\mathbf{a})] \star \Theta\right\} \\
\Rightarrow & \{\text{use side conditions 3 and 4}\} \\
 & \{\Phi\} \cdot (\mathbf{t})\left\{\left(\bigvee_{i=1}^{m} \exists \mathbf{v}_i.\Upsilon'_i[\mathbf{a}\backslash I(\mathbf{a})]\right) \star \Theta\right\} \\
\Rightarrow & \{\text{distribution of } \vee \text{ and } \star\} \\
 & \{\Phi\} \cdot (\mathbf{t})\left\{\bigvee_{i=1}^{m} (\exists \mathbf{v}_i.\Upsilon_i[\mathbf{a}\backslash I(\mathbf{a})] \star \Theta)\right\}
\end{array}$$

To finish, we need to prove (40). Clearly we have

$$\Upsilon_k[\mathbf{u}_k\backslash I(\mathbf{u}_k)] \;\Rightarrow\; \exists \mathbf{u_k}.\Upsilon_k$$

Then by universal generalisation we have

$$\forall \mathbf{a}.(\Upsilon_k[\mathbf{u}_k\backslash I(\mathbf{u}_k)] \;\Rightarrow\; \exists \mathbf{u_k}.\Upsilon_k)$$

and then instantiating $\mathbf{a}$ with $I(\mathbf{a})$ gives us

$$\Upsilon_k[\mathbf{u}_k\backslash I(\mathbf{u}_k)][\mathbf{a}\backslash I(\mathbf{a})] \;\Rightarrow\; (\exists \mathbf{u_k}.\Upsilon_k)[\mathbf{a}\backslash I(\mathbf{a})]$$

Thus we will be done if we can show that

$$\Upsilon_k[\mathbf{u}_k\backslash I(\mathbf{u}_k)][\mathbf{a}\backslash I(\mathbf{a})] \;\Leftrightarrow\; \Upsilon_k[\mathbf{u}_k, \mathbf{a}\backslash I(\mathbf{u}_k, \mathbf{a})]$$

and this follows from side conditions 6 and 7.

G Extended rules for $\vdash_{\textbf{find-tr}}$

FindGuidedUnfold

$$\frac{\begin{array}{c}\mathit{purify}(\mathit{closure}(\Psi_i[\mathbf{v}_i\backslash\mathbf{w}_i][\mathbf{x}\backslash\mathbf{e}] \star P)) \vdash_{SMT} \mathit{false} \text{ for all } i \neq k \\ \Pi, \mathrm{X}(\mathbf{v}) \Leftrightarrow (\exists \mathbf{v}_1.\Psi_1) \vee \cdots \vee (\exists \mathbf{v}_n.\Psi_n) : \Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{x}\backslash\mathbf{e}] \star P \vdash^{\mathbf{G'}}_{\text{find-tr}} \exists \mathbf{y}.\, e \mapsto B(\cdot) \star R^{pure}\end{array}}{\Pi, \mathrm{X}(\mathbf{v}) \Leftrightarrow (\exists \mathbf{v}_1.\Psi_1) \vee \cdots \vee (\exists \mathbf{v}_n.\Psi_n) : \mathrm{X}(\mathbf{e}) \star P \vdash^{\mathbf{G}}_{\text{find-tr}} \begin{array}{l}\exists \mathbf{y}, \mathbf{w_k}.\, e \mapsto B(\cdot) \\ \star\ \mathit{purify}(\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{x}\backslash\mathbf{e}]) \star R^{pure}\end{array}}$$

$$\mathbf{w}_1, \ldots, \mathbf{w}_n \text{ fresh}; \qquad \mathbf{G} = \mathsf{unfold}\ X(\mathbf{e^?}), \mathbf{G'}; \qquad e_i^? \in \{e_i, ?\}$$

FindGuidedSplit

$$\frac{\begin{array}{c}\mathsf{recdef}\ \mathrm{L}(s,t,\alpha) := \Phi \in \mathrm{LsegDefn}(n, \mathbf{v}, [\mathscr{C}_1, \ldots, \mathscr{C}_K], [A_1, \ldots, A_N], [E_1, \ldots, E_k]) \\ P \vdash^I \exists c_1, \ldots, c_k\ .\ (q_1, \ldots, q_k) \in \hat{e}_\gamma \star \Theta \\ S = \begin{pmatrix} \mathrm{L}(\hat{e}, s, \alpha) \\ \star\ s \mapsto \mathscr{C}_1, \ldots, \mathscr{C}_K, n \\ \star\ A_1 \star \cdots \star A_N \\ \star\ \mathrm{L}(n, \hat{e}', \beta) \\ \star\ \hat{e}_\gamma = \alpha \cup \{(e_1, \ldots, e_k)\} \cup \beta \\ \star\ E_1 = e_1 \star \cdots \star E_k = e_k \end{pmatrix} \\ \Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow \Phi : S[s, n, \mathbf{v}\backslash\mathbf{w}] \star P \vdash^{\mathbf{G'}}_{\text{find-tr}} \exists \mathbf{y}.\, e \mapsto B(\cdot) \star R^{pure}\end{array}}{\Pi, \mathrm{L}(\mathbf{x}) \Leftrightarrow \Phi : \mathrm{L}(\hat{e}, \hat{e}', \hat{e}_\gamma) \star P \vdash^{\mathbf{G}}_{\text{find-tr}} \exists \mathbf{y}, \alpha, \beta, \mathbf{w}.\, e \mapsto B(\cdot) \star \mathit{purify}(S[s, n, \mathbf{v}\backslash\mathbf{w}]) \star R^{pure}}$$

$$\begin{array}{l}\alpha, \beta, \mathbf{w} \text{ fresh},\ c_1, \ldots, c_k \text{ fresh}, \\ \mathbf{G} = \mathsf{split}\ X\ e\ (\mathbf{p^?}), \mathbf{G'} \\ \text{each } p_i \text{ is a value expression or } \_, \\ \text{each } q_i \text{ is } \begin{cases} c_i & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}, \\ \text{each } e_i \text{ is } \begin{cases} I(c_i) & \text{if } p_i \text{ is ?} \\ p_i & \text{otherwise} \end{cases}\end{array}$$

H Soundness of Rules for $\vdash_{\textbf{find-tr}}$

H.1 Soundness of Find

Find

$$\frac{P \vdash_{SMT} e'_A = e_A + o}{\Pi : P \ \star\ e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, B, \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \vdash^{\varepsilon}_{\text{find-tr}} e'_A \mapsto B}$$

We know by soundness of $\vdash_{SMT}$ that

$$P \Rightarrow e'_A = e_A + o \tag{42}$$

and need to show that $\Pi \models P \ \star\ e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, B, \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \Rightarrow e'_A \mapsto B \ \star\ \Upsilon$ for some appropriate Υ.

$$\begin{array}{ll} & P \ \star\ e_A \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, B, \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \\ \Rightarrow & \{\ (\mapsto\text{-Group}), (\star\text{-Monotonicity})\ \} \\ & P \ \star\ e_A \mapsto \mathscr{C}_0 \ \star\ \ldots\ \star\ e_A + n \mapsto \mathscr{C}_n \\ \Rightarrow & \{\ \text{by (SplitPureRight), } (\star\text{-Monotonicity}), \text{ and (42)}\ \} \\ & P \ \star\ e'_A = e_A + o \ \star\ e_A \mapsto \mathscr{C}_0 \ \star\ \ldots\ \star\ e_A + n \mapsto \mathscr{C}_n \\ \Rightarrow & \{\ \text{by equational reasoning, (SplitPureLeft), } (\star\text{-Monotonicity})\ \} \\ & P \ \star\ e_A \mapsto \mathscr{C}_0 \ \star\ \ldots\ \star\ e_A + o - 1 \mapsto \mathscr{C}_{o-1} \ \star\ e \mapsto B \ \star \\ & \quad \star\ e_A + o + 1 \mapsto \mathscr{C}_{o+1} \ \star\ \ldots\ \star\ e_A + n \mapsto \mathscr{C}_n \\ \Rightarrow & \{\ (\star\text{-Commutative})\ \} \end{array}$$

$$e'_A \mapsto B \star \Upsilon$$

where $\Upsilon \equiv P \star e_A \mapsto \mathscr{C}_0 \star \ldots \star e_A{+}o{-}1 \mapsto \mathscr{C}_{o-1} \star e_A{+}o{+}1 \mapsto \mathscr{C}_{o+2} \star \ldots \star e_A{+}n \mapsto \mathscr{C}_n$.

H.2 Soundness of FindUnfold

$$\frac{\begin{array}{c} \mathit{purify}(\mathit{closure}(\Psi_i[\mathbf{v}_i\backslash\mathbf{w}_i][\mathbf{v}\backslash\mathbf{E}] \star P)) \vdash_{SMT} \mathit{false} \text{ for all } i \neq k \\ \Pi, \mathrm{X}(\mathbf{v}) \Leftrightarrow \exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n : \Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}] \star P \vdash^{\varepsilon}_{\text{find-tr}} \exists \mathbf{y}.\, e_A \mapsto B \star R^{pure} \end{array}}{\begin{array}{l} \Pi, \mathrm{X}(\mathbf{v}) \Leftrightarrow \exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n : \mathrm{X}(\mathbf{E}) \star P \vdash^{\varepsilon}_{\text{find-tr}} \exists \mathbf{y}, \mathbf{w_k}.\, e_A \mapsto B \\ \qquad \star\ \mathit{purify}(\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}]) \\ \qquad \star R^{pure} \\ \mathbf{w}_1, \ldots, \mathbf{w}_n \text{ fresh} \end{array}}$$

Note that the first hypothesis of the rule is just there to make the rule deterministic. By the first assumption and soundness of $\vdash_{SMT}$ we know that

$$\mathit{purify}(\mathit{closure}(\Psi_i[\mathbf{v}_i\backslash\mathbf{w}_i][\mathbf{v}\backslash\mathbf{E}] \star P)) \Leftrightarrow \mathit{false} \qquad \text{for all } i \neq k \tag{43}$$

The following lemma can be simply derived from the laws for *puriy* and *closure*:

Lemma 11 *If* $\mathit{purify}(\mathit{closure}(A)) \Leftrightarrow \mathit{false}$ *then* $A \Leftrightarrow \mathit{false}$.

From Lemma 11 and (43) it follows that $\Psi_i[\mathbf{v}_i\backslash\mathbf{w}_i][\mathbf{v}\backslash\mathbf{E}] \star P \Leftrightarrow \mathit{false}$ for all $i \neq k$ and thus

$$\exists \mathbf{v_i}.\, \Psi_i[\mathbf{v}\backslash\mathbf{E}] \star P \Leftrightarrow \mathit{false} \qquad \text{for all } i \neq k \tag{44}$$

By the soundness of the second assumption (using induction hypothesis) we know that

$$\Pi, X(\mathbf{v}) \Rightarrow \exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n \ \models\ \Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}] \star P \Leftrightarrow \exists \mathbf{y}.\, e_A \mapsto B \star R^{pure} \star \Upsilon' \tag{45}$$

for some Υ' and need to show that

$$\begin{array}{l} \mathrm{X}(\mathbf{v}) \Leftrightarrow \exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n \ \models \\ \qquad \mathrm{X}(\mathbf{E}) \star P \Leftrightarrow \exists \mathbf{y}, \mathbf{w_k}.\, e_A \mapsto B(\cdot) \star \mathit{purify}(\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}]) \star R^{pure} \star \Upsilon \end{array}$$

for some appropriate Υ. We reason as follows:

$$\begin{array}{ll} & \mathrm{X}(\mathbf{E}) \star P \\ \Rightarrow & \{(\textsc{PredLeft})\} \\ & (\exists \mathbf{v}_1.\Psi_1 \vee \cdots \vee \exists \mathbf{v}_n.\Psi_n)[\mathbf{v}\backslash\mathbf{E}] \star P \\ \Rightarrow & \{ \text{ by the } \star \text{ distribution rules } \vee \text{ and } \exists \text{ and } (\star\text{-}\textsc{Monotonicity}) \} \\ & (\exists \mathbf{v}_1.\Psi_1[\mathbf{v}\backslash\mathbf{E}] \star P) \vee \cdots \vee (\exists \mathbf{v}_n.\Psi_n[\mathbf{v}\backslash\mathbf{E}] \star P) \\ \Rightarrow & \{ \text{ by (44)} \} \\ & \exists \mathbf{v}_k.\Psi_k[\mathbf{v}\backslash\mathbf{E}] \star P \end{array}$$

It therefore remains to show

$$\exists \mathbf{v}_k.\Psi_k[\mathbf{v}\backslash\mathbf{E}] \star P \Rightarrow \exists \mathbf{y}, \mathbf{w_k}.\, e_A \mapsto B \star \mathit{purify}(\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}]) \star R^{pure} \star \Upsilon \tag{46}$$

for some Υ. For the remainder, first note that the following logical rule (which is standard):

$$\frac{\forall\exists \quad \forall x.\, A \Rightarrow B}{(\exists x.\, A) \Rightarrow (\exists x.\, B)}$$

To prove (46) by the $(\forall\exists)$ rule it thus suffices to show (as $\mathbf{w_k}$ are fresh):

$$\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}] \star P \Rightarrow \exists \mathbf{y}.\, e_A \mapsto B \star \mathit{purify}(\Psi_k[\mathbf{v}_k\backslash\mathbf{w}_k][\mathbf{v}\backslash\mathbf{E}]) \star R^{pure} \star \Upsilon \tag{47}$$

choosing Υ to be Υ' this follows from (45).

H.3 Soundness of FindSplit

FindSplit

$$\frac{\begin{array}{c}\mathsf{recdef}\,\mathrm{L}(s,t,\alpha) := Q \in \mathrm{LsegDefn}(n,\mathbf{v},[\mathscr{C}_1,\ldots,\mathscr{C}_K],[A_1,\ldots,A_N],[E_1,\ldots,E_k]) \\ P \vdash^I \exists u_1,\ldots,u_k\,.\,(u_1,\ldots,u_k) \in \hat{e}_\gamma \star \Theta \\ S = \begin{pmatrix} \mathrm{L}(\hat{e},s,\alpha) \\ \star\, s \mapsto \mathscr{C}_1,\ldots,\mathscr{C}_K,n \\ \star\, A_1 \star \cdots \star A_N \\ \star\, \mathrm{L}(n,\hat{e}',\beta) \\ \star\, \hat{e}_\gamma = \alpha \cup \{(I(u_1),\ldots,I(u_k))\} \cup \beta \\ \star\, E_1 = I(u_1) \star \cdots \star E_k = I(u_k) \end{pmatrix} \\ \Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow Q : S[s,n,\mathbf{v}\backslash\mathbf{w}] \star P \vdash^{\varepsilon}_{\text{find-tr}} \exists \mathbf{y}.\, e_A \mapsto B \star R^{pure}\end{array}}{\Pi, \mathrm{L}(s,t,\alpha) \Leftrightarrow Q : \mathrm{L}(\hat{e},\hat{e}',\hat{e}_\gamma) \star P \vdash_{\text{find-tr}} \exists \mathbf{y},\mathbf{w},\alpha,\beta.\, e_A \mapsto B \star \mathit{purify}(S[s,n,\mathbf{v}\backslash\mathbf{w}]) \star R^{pure}}$$

$\alpha, \beta, \mathbf{w}$ fresh, $u_1,\ldots,u_k$ fresh

By the first assumption and soundness of $\vdash^I$ we know that

$$P \Rightarrow (I(u_1),\ldots,I(u_k)) \in \hat{e}_\gamma \star \Theta \tag{48}$$

From (48) we get $P \Rightarrow (I(u_1),\ldots,I(u_k)) \in \hat{e}_\gamma \star \mathit{true}$ and thus by ($\star$-SplitPureRight)

$$P \Rightarrow P \star (I(u_1),\ldots,I(u_k)) \in \hat{e}_\gamma \tag{49}$$

The second assumption by induction yields:

$$S[s,n,\mathbf{v}\backslash\mathbf{w}] \star P \Rightarrow \exists \mathbf{y}.\, e_A \mapsto B \star R^{pure} \star \Upsilon \tag{50}$$

for some Υ. First we reason as follows:

$$\begin{array}{ll} & \mathrm{L}(\hat{e},\hat{e}',\hat{e}_\gamma) \star P \\ \Rightarrow & \{\ \text{(by (49), ($\star$-Monotonicity),($\star$-Commutativity)}\ \} \\ & \mathrm{L}(\hat{e},\hat{e}',\hat{e}_\gamma) \star (I(u_1),\ldots,I(u_k)) \in \hat{e}_\gamma \star P \\ \Rightarrow & \{\ \text{by (Split) and ($\star$-Monotonicity)}\ \} \\ & \exists s,n,\mathbf{v},\alpha,\beta.S \star P \end{array}$$

Using ($\forall\exists$) again it just remains to show (note that α and β are already fresh by assumption and thus need not be substituted):

$$S[s,n,\mathbf{v}\backslash\mathbf{w}] \star P \Rightarrow \exists \mathbf{y}.\, e_A \mapsto B \star \mathit{purify}(S[s,n,\mathbf{v}\backslash\mathbf{w}]) \star R^{pure} \star \Upsilon \tag{51}$$

for some Υ. By (Purify) and ($\star$-Monotonicity) it suffices to show

$$S[s,n,\mathbf{v}\backslash\mathbf{w}] \star \mathit{purify}(S[s,n,\mathbf{v}\backslash\mathbf{w}]) \star P \Rightarrow \exists \mathbf{y}.\, e \mapsto B \star \mathit{purify}(S[s,n,\mathbf{v}\backslash\mathbf{w}]) \star R^{pure} \star \Upsilon \tag{52}$$

for some Υ and by ($\star$-Commutativity) and ($\star$-Monotonicity) this holds if we can prove that

$$S[s,n,\mathbf{v}\backslash\mathbf{w}] \star P \Rightarrow \exists \mathbf{y}.\, e_A \mapsto B \ \star R^{pure} \ \star \Upsilon \tag{53}$$

for some Υ which follows from (50).

I Detailed steps of the entailment prover

Here we show how the unfolded strong specifications can be folded up into the weak variations, as performed in *main* and discussed in Section 6.5. The proof starts with symbolic state obtained at the point just before the *unfold* ghost statement executes.

$\{\$ListLibraryStrong(lookupL, addPairL, disposeL, createL) \star res \mapsto \_\}$

ghost unfold $\$ListLibraryStrong(?,?,?,?)$;

UNFOLD

$$\left\{ \begin{array}{l} lookupL \mapsto \ldots \star disposeL \mapsto \ldots \star createL \mapsto \ldots \\ \star\ addPairL \mapsto \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \star\ res \mapsto \_ \end{array} \right\}$$

ghost fold $\$ListLibraryWeak(?,?,?,?)$;

FOLD (see section I.1 below. Finds frame $res \mapsto \_$)

$\{\$ListLibraryWeak(lookupL, addPairL, createL, disposeL) \star res \mapsto \_\}$

I.1 Entailment steps for the (GHOSTFOLD) rule

For the fold, using the defintion of $\$ListLibraryWeak$ we are trying to prove

$$\begin{array}{l} lookupL \mapsto \ldots \star disposeL \mapsto \ldots \star createL \mapsto \ldots \\ \star\ addPairL \mapsto \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \star\ res \mapsto \_ \\ \vdash \\ \exists a_0, a_1, a_2, a_3. \\ a_0 \mapsto \ldots \star a_1 \mapsto \ldots \star\ a_2 \mapsto \ldots \\ \star\ a_3 \mapsto \forall al, key, value. \\ \left\{ \begin{array}{l} \exists \kappa.\ \$AssocListH(al;\kappa) \\ \star\ \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \exists \kappa.\ \$AssocListH(al;\kappa) \right\} \end{array}$$

Applied (INSTMATCHADDR) [Instantiating a_3 with addPairL]

$$\begin{array}{l} lookupL \mapsto \ldots \star disposeL \mapsto \ldots \star createL \mapsto \ldots \\ \star\ addPairL \mapsto \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \star\ res \mapsto \_ \\ \vdash \\ \exists a_0, a_1, a_2. \\ a_0 \mapsto \ldots \star a_1 \mapsto \ldots \star\ a_2 \mapsto \ldots \\ \star\ addPairL \mapsto \forall al, key, value. \\ \left\{ \begin{array}{l} \exists \kappa.\ \$AssocListH(al;\kappa) \\ \star\ \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \exists \kappa.\ \$AssocListH(al;\kappa) \right\} \end{array}$$

Applied (CANCELPTTRIPLE) (see section I.2 below).

$$\begin{array}{l} lookupL \mapsto \ldots \star disposeL \mapsto \ldots \star createL \mapsto \ldots \star\ res \mapsto \_ \\ \vdash \\ \exists a_0, a_1, a_2.\ a_0 \mapsto \ldots \star a_1 \mapsto \ldots \star a_2 \mapsto \ldots \end{array}$$

Applied (INSTMATCHADDR) and (CANCELPTTRIPLE) in a similar way for instantiations $[a_2 := createL, a_1 := disposeL, a_0 := lookupL]$

$$res \mapsto \_ \quad \vdash \quad \mathsf{emp}$$

Applying (PURE) to finish, we have found the frame $res \mapsto \_$.

I.2 Entailment steps for the application of (CancelPtTriple) with addPairL

For this, we need to prove

$$\begin{array}{l} \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \vdash \\ \forall al, key, value. \\ \left\{ \begin{array}{l} \exists \kappa.\ \$AssocListH(al;\kappa) \\ \star\ \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \exists \kappa.\ \$AssocListH(al;\kappa) \right\} \end{array}$$

Applied (RemoveForallRight)

$$\begin{array}{l} \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \vdash \\ \left\{ \begin{array}{l} \exists \kappa.\ \$AssocListH(al_0;\kappa) \\ \star\ \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ \begin{array}{l} \exists \kappa. \\ \$AssocListH(al_0;\kappa) \end{array} \right\} \end{array}$$

Applied (ExistsPre)

$$\begin{array}{l} \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \vdash \\ \left\{ \begin{array}{l} \$AssocListH(al_0;\kappa_0) \\ \star\ \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ \begin{array}{l} \exists \kappa. \\ \$AssocListH(al_0;\kappa) \end{array} \right\} \end{array}$$

Applied (TripleEnt) (see the two separate parts below)

I.2.1 Trying to relate preconditions (first premise).

Trying to prove

$$\begin{array}{l} \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star \$Rel(key, value) \end{array} \right\} \cdot(al, key, value) \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\} \\ \vdash_{\text{find-post}} \\ \left\{ \begin{array}{l} \$AssocListH(al_0;\kappa_0) \\ \star\ \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ ? \right\} \end{array}$$

Applied (InstParam)

$$\begin{array}{l} \forall \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al_0;\kappa) \\ \star \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ \$AssocListH(al_0; \{key_0\} \cup \kappa) \right\} \\ \vdash_{\text{find-post}} \\ \left\{ \begin{array}{l} \$AssocListH(al_0;\kappa_0) \\ \star\ \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ ? \right\} \end{array}$$

Applied (InferSpecForCall) (see steps below)

$$\begin{array}{l} \forall \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al_0;\kappa) \\ \star \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ \$AssocListH(al_0; \{key_0\} \cup \kappa) \right\} \\ \vdash_{\text{find-post}} \\ \left\{ \begin{array}{l} \$AssocListH(al_0;\kappa_0) \\ \star\ \$Rel(key_0, value_0) \end{array} \right\} \cdot(al_0, key_0, value_0) \ \left\{ \$AssocListH(al_0; \{key_0\} \cup \kappa_0) \right\} \end{array}$$

This completes proof of first premise, and found postcondition:

$$\$AssocListH(al_0; \{key_0\} \cup \kappa_0)$$

Showing premise for (InferSpecForCall). Trying to prove

$$\begin{array}{r} \$AssocListH(al_0; \kappa_0) \\ \star\ \$Rel(key_0, value_0) \end{array} \quad \vdash \quad \begin{array}{l} \exists \kappa.\ \ \$AssocListH(al_0; \kappa) \\ \quad \star\ \$Rel(key_0, value_0) \end{array}$$

Applied (InstMatchArg) $[\kappa := \kappa_0]$, (CancelPred)

$$\$Rel(key_0, value_0) \quad \vdash \quad \$Rel(key_0, value_0)$$

Applied (CancelPred)

$$\mathsf{emp} \quad \vdash \quad \mathsf{emp}$$

(Pure) completes the entailment. There is no frame left over, so the generated postcondition is simply the postcondition from the LHS, after substituting with the discovered instantiation $[\kappa := \kappa_0]$:

$$\$AssocListH(al_0; \{key_0\} \cup \kappa_0)$$

I.2.2 Trying to relate postconditions (second premise).

Trying to prove

$$\$AssocListH(al_0; \{key_0\} \cup \kappa_0) \quad \vdash \quad \exists \kappa_1.\$AssocListH(al_0; \kappa_1)$$

Applied (InstMatchArg) $[\kappa_1 := \{key_0\} \cup \kappa_0]$, (CancelPred)

$$\mathsf{emp} \quad \vdash \quad \mathsf{emp}$$

(Pure) completes postconditions.